A verified compiler for Handel-C

Juan Ignacio Perna

Ph.D. Thesis
This thesis is submitted in partial fulfilment of the requirements for the degree of Doctor of Philosophy.

The University of York
High Integrity Systems Engineering Group
Department of Computer Science
United Kingdom

November 2009

Abstract

The recent popularity of Field Programmable Gate Array (FPGA) technology has made the synthesis of Hardware Description Language (HDL) programs into FPGAs a very attractive topic for research. In particular, the correctness in the synthesis of an FPGA programming file from a source HDL program has gained significant relevance in the context of safety or mission-critical systems.

The results presented here are part of a research project aiming at producing a *verified compiler* for the Handel-C language. Handel-C is a high level HDL based on the syntax of the C language extended with constructs to deal with parallel behaviour and process communications based on CSP.

Given the complexity of designing a provably correct compiler for a language like Handel-C, we have adopted the *algebraic approach* to compilation as it offers an elegant solution to this problem. The idea behind algebraic compilation is to create a sound reasoning framework in which the a formal model of the source Handel-C program can be embedded and refined into a formal abstraction of the target hardware. As the algebraic rules used to compile the program are proven to preserve the semantics, the correctness of the entire compilation process (i.e., semantic equivalence between source and target programs) can be argued *by construction*, considering each programming construct in isolation, rather than trying to assert the correctness of the compilation in a single step.

Regarding hardware synthesis, the algebraic approach has already been applied to subsets of Occam and Verilog. Our work builds on some ideas from these works but focuses on the more complex timing model imposed by Handel-C. Moreover, our work covers features like shared variables, multi-way communications and priorities which, to our knowledge, have never been addressed within the framework of algebraic compilation.

Finally, one characteristic of the algebraic approach is that the basic reduction laws in the reasoning framework are postulated as axioms. As an invalid axiom would allow us to prove invalid results (up to the extent of being able to prove a false theorem) we are also concerned about the consistency of the basic postulates in our theory. We addressed this by providing denotational semantics for Handel-C and its reasoning extensions in the context of the Unifying Theories of Programming (UTP). Our UTP denotational semantics not only provided a model for our theory (hence, proving its consistency) but also allowed us to prove all the axioms in the compilation framework.

Acknowledgements

I would like to express my sincere thanks to my supervisor, Prof. Jim Woodcock, for his comments and support as the ideas in this thesis evolved, and for his encouragement and help when they were not evolving so well. Throughout these three years I have been working under Jim's supervision I have not only got to recognise and admire his technical skills, vast knowledge and insights about the UTP theory, but also his gentle guidance that kept me on track with the research while allowing me to explore and learn in my own way.

Most of the work on the compilation aspects of this thesis was developed under Augusto Sampaio's supervision while visiting Universidade Federal de Pernambuco (UFPE) in Recife, Brazil. During this time, I also had many fruitful discussions Juliano Iyoda about hardware generation and theorem proving. Working with Augusto and Juliano was an enriching experience that greatly contributed to this thesis completion and for that, my gratitude will always be with them. Besides the technical aspects, I will always be grateful to Augusto, Juliano and Adalberto Farias for making me feel I was among friends while at UFPE.

My sincere thanks to Leo Freitas for enduring the reading of early drafts of this thesis and for his valuable and thorough feedback. His comments have greatly improved the quality of the final version of this document. I also want to thank Ian Gray, Matthew Naylor and Neil Audsley for their advice and helpfulness regarding Handel-C, hardware synthesis and FPGAs.

To my good friends Leo, Jan Tobias Mühlberg and Nada Seva. The three of them filled my days with interesting discussions, enjoyable (grumpy, but still lovely, in the case of Tobias) moments and great support during the up-and-downs of the PhD. To me, they are the kind of friends that make one's life colourful and worth living. During this three years in York I also have crossed paths with people I am honoured and glad of calling friends (in no particular order): Tom Lampert, Pierre Andrews, Silvia Quarteroni, Silvia Ursprung, Bernadete Martinez-Hernandez , Estefania de Vasconcellos Guimaraes, Enda Ridge, Giuseppe Montano, Frank Zeyda, Sergio & Anick Mena, Malihe Tabatabaie, Burcu Can, Richard Ribero, Marek & Marta Grzes, Ken Wolfe, Rana Tayara, Silvana Robonne and Jennifer Woodland. Also to my good friends: Noelia Sorokin, Evelyn Suarez, Alejandro Spinola and Fernando Rigoni; with whom I feel at home every time I go back to Argentina.

I have always thought that people that do Computer Science need, at least, another (less "computer-based") activity where we can express things that cannot be done with a keyboard and a mouse... In my case, this means Argentinian Tango. It was through tango that I met Nada, and later on, her husband Srdjan Seva. I am grateful to both of them for the joint teaching, the long hours dancing together (even with Srdjan!), for showing me that there is tango other than

Pugliese, and for their patience when trying to improve my milonga. I also want to thank my first tango teachers and friends, Graciela and Osvaldo, with whom I have shared so many wonderful experiences.

Even though I consider myself lucky for the great friends I have made during the PhD, my life in York would not have been complete if Sarah was not part of it. She not only stoically endured my boring technical explanations and my absent-minded moments when the solution to a problem from the PhD came in the middle of a conversation. She also filled my days with understanding love, laughs, tenderness, salsa dancing, and happiness. To her all my love and gratitude for holding my hand during the PhD and for making me a better person.

Finally, to my parents and brothers, for their unconditional love, encouragement and support throughout my life.

Declaration

This thesis has not previously been accepted in substance for any degree and is not being concurrently submitted in candidature for any degree other than Doctor of Philosophy of the University of York. This thesis is the result of my own investigations, except where otherwise stated. Other sources are acknowledged by explicit references.

I hereby give consent for my thesis, if accepted, to be made available for photocopying and for inter-library loan, and for the title and summary to be made available to outside organisations.

Signed ..... (candidate)

Date .....

Contents

Chapter 1

Introduction

Software development is notoriously error prone [Leveson and Turner 1993; Lann 1997], to the extent that malfunctioning and incorrect programs have become a part of our everyday lives. In particular, there are numerous examples of cases where software errors brought enormous financial losses to businesses [Newman 2002] and society in general [MacKenzie 2001]. As a consequence, various approaches have been used to make software more reliable.

One of the most widely used approaches to ensure software correctness is testing [Gelperin and Hetzel 1988; Beizer 1990; Myers and Sandler 2004; Hierons et al. 2009]. It consists of running software on input data that is believed to be representative of what can be encountered in practice, and comparing the actual behaviour of programs to the expected behaviour. Unfortunately, the degree of testing that can be performed in most of the cases cannot guarantee the required level of reliability. The main reason for the insufficient effectiveness of software testing is the large number of possible execution paths in programs and discontinuous behaviour of software systems. In particular, testing on just some of the possible inputs is usually not enough because software is discrete, and knowing the behaviour on some input may provide almost no assurance on the correctness of the behaviour on similar input.

On the other hand, the required level of software reliability can potentially be achieved with formal methods [Bergstra and Klop 1985; Pfleeger and Hatton 1997; Davis 2005]. Contrary to testing, formal methods provide a mathematical model of a system that allows the properties and behaviour of a given design to be predicted through mathematical/mechanical reasoning. Thus formal methods can potentially move the construction and validation of software (away from experiment and adjustment) towards prediction and calculation. The development of reliable systems using formal methods comprises many stages from the original requirements down to the hardware in which programs will run. Many methods, theories, techniques and tools have been developed to deal with adjacent stages of the development process. For example correct generation of programs from specifications, the translation of programs into machine code and sometimes even to a hardware implementation level.

Programming languages constitute a fundamental step in this process, aiming at making programming easier and less error prone. The price of this is the need to use a compiler or an interpreter, which is another, usually complex (and equally error-prone) piece of software. Reliability

of interpreters is beyond the scope of this work, so we will only refer to compilation. When we compile a program, we would like to be sure that the compiled program does exactly what the source one describes. Ideally, we would like to be able to show, by formal arguments, that the output produced by a compiler is equivalent to (or a refinement of) the source code.

The question of compiler correctness, as well as the rigorous specification of programming languages, is a big issue for software developers and users. An error introduced during the compilation process due to either incorrect implementation of the compiler or a misinterpretation of the programming language specification can cause severe problems, especially if it occurs in a safety-critical application. Moreover, all the effort invested at previous development stages to guarantee the software correct can be just wasted by a faulty compiler.

Compiler construction theory is one foundation for high-level languages and, as such, has been subject of study for many years [Aho et al. 1986; Aho and Ullman 1972; Davie and Morrison 1982; Tremblay and Sorenson 1985]. Within this field, one of the most important goals is to certify the correctness of the implementation of a given programming language compiler. In this context, by correct we mean that the compiler must be proved to preserve the behaviour (or semantics) of the source when it is translated into a target program. To achieve compiler correctness, several formal approaches have been devised and explored. The most relevant ones that can be mentioned are based on different formalisms, such as weakest precondition calculus [Dijkstra 1976], higher-order logics [Gordon and Melham 1993], action semantics [Mosses 1996] and algebraic reduction rules [Hoare 1985].

On the other hand, the performance and major industrial adoption [Hartenstein 1997; Raje 2004; Wright and Arens 2005; Joost and Salomon 2005] of Field Programmable Gate Arrays (FPGA) has made the compilation of Hardware Description Languages (HDLs) a very attractive topic for research. FPGAs are now present not only in every-day-use devices but also in controllers for spacecraft, defence systems, high-speed trains, medical systems and so on. In this particular context of mission- or life-critical systems, it is fundamental to ensure, not only the correctness when specifying the system at HDL level (e.g., validation against requirements and verification/proof of desired properties) but also that the generated hardware is semantically equivalent to the HDL level program (i.e., the hardware behaves in the same way, and preserves all the properties that were valid at the original specification-level program).

This work addresses the issue of compiler correctness for HDLs. By *correctness* we mean a compiler that is guaranteed to produce a translation (hardware components in this particular case) that is semantically an improvement (i.e., a refinement) of the source specification.

1.1 Alternatives towards program and compiler correctness

Software testing approaches do not provide the level of confidence required in most safety-critical environments [Butler and Finelli 1991]. Formal methods involve the use of mathematical techniques for inferring the properties and behaviour of programs, thus potentially providing greater levels of confidence in software correctness.

This section describes the major formal approaches to establishing the correctness of the com-

pilation process or some aspects of it. In particular, *formal verification*, *proof-carrying code* and the *algebraic compilation* methods are covered. The list of methodologies described in this section is certainly not comprehensive, it is just presenting the approaches that, to our understanding, are the most promising ideas in the field.

1.1.1 Formal verification

Formal verification consists of formally establishing the correctness of a program P by providing: (i) a definition of what we mean by *correct* and (ii) a method for establishing that a program P satisfies the correctness criteria for all possible inputs. As compilers are themselves programs, formal verification can be applied to them to establish their correctness. The inputs of a compiler are other programs and its outputs are the same programs converted into a different form.

Compilers are usually written in high-level programming languages, which cannot be executed directly. This implies that, before being executed, compilers have to be transformed into binary machine code (i.e., be compiled themselves), and the correctness of the machine code should be proven with respect to the compilation specification. It is the object code that we would finally like to be correct.

Because of its complexity, the task of proving the correctness of the object code is typically left out in most works concerned with compiler verification. As result, verification of even a very simple compiler for a very simple source and target programming languages can already be too difficult, as establishing the correctness of commercial programming languages compilers' code is prohibitively expensive.

There are numerous examples of partial verification of the compilation of sequential [Polak 1981], object oriented (Java) [Berghofer and Strecker 2003; Klein and Nipkow 2003; Goos 2002], concurrent [Young 1988; 1989], high-level assembly [Moore 1989], compiler back-ends [Oliva 1994] and functional [Goerigk et al. 1996] languages. Furthermore, there has been research towards the verification of optimisations in a general framework [Goerigk 2002] and for embedded systems [Glesner et al. 2002]. Unfortunately, in most of these works, proving the correctness of the final machine code is left out, or is informally addressed. In this sense, one of the most remarkable results in compiler verification has been achieved in Verifix [Goerigk et al. 1996], which aimed at the construction of provably correct compilers for ComLisp, a subset of Common Lisp.

On the other hand, the most remarkable work in this context is a fully verified compiler from ComLisp to binary transputer [Barron 1978] code was implemented using ComLisp itself as the language [Dold and Vialard 2001; Goerigk and Simon 1999; Hoffmann 1998]. The compilation is carried out through transformations into a series of intermediate languages specified by means of operational semantics. On top of this, the PVS [Owre et al. 2001] system was chosen to support verification of the compilation specification and the construction process of the compiler in the source language. The ComLisp implementation was compiled with some arbitrary Common Lisp compiler and then applied to itself (i.e., compiler bootstrapping). Thus, the final executable code was generated according to the compilation specification and its compliance was then syntactically checked [Hoffmann 1998]. Despite quite simple source language and limited compiler complexity, "the size and complexity of the verification task in constructing a correct compiler is immense"

[Dold and Vialard 2001]. Even though it took several years to create this compiler, ComLisp can be regarded as state-of-the-art technology, and its verified compiler as one of the most successful results in the field of compiler correctness verification.

On the other hand, there have been successful cases of correct-by-construction compilers – the compiler is built correctly rather than producing a compiler and then trying to verify it – for Java [Stark et al. 2001] and a C-like programming subset [Stepney 1993; 1998]. In the case of the correct compiler for Java, Generalised State Machines are used as the underlying formalism and all constructs are given semantics in those terms. A correct compiler capable of addressing most constructs (with exception of threads) is derived from the operational semantics for the language and its correctness is argued informally. On the other hand, the work on C-like languages uses Z [Woodcock and Davies 1996] as the semantic domain for both operational and denotational semantics for the language. Again, a correct compiler is derived from the operational semantics and its correctness is formally argued first manually and later on by means of an embedding in the PVS proof system [Stepney et al. 1991]. The main limitation of this work lies in the fact that it assumes termination (hence, the semantic framework can only reason about partial correctness). This is a limitation inherited from Z that does not provide a least fixed point operator for schemas, pre-empting the possibility of reasoning about the termination of recursion.

1.1.2 Proof-carrying code

Proof-carrying code (PCC) [Necula 1997; Necula and Lee 1998; Colby et al. 2000] is a completely different and independent approach compared to formal verification. Instead of establishing a correspondence between the source and compiled programs, proof-carrying code is used to certify safety properties (i.e., properties ensuring certain undesirable behaviour is never allowed) on the target (compiled) program. PCC does not remove the necessity of verification, but suggests the program producers supply with the program an explicit proof of the program properties. Thus, the consumers of the program can be sure of the safety of a program obtained from an untrusted person without any verification or run-time check on their side.

The PCC approach consists of: (i) defining a safety policy that precisely specifies the conditions that the execution of a given program should meet in order to be considered safe; (ii) generating a proof that the program adheres to the safety policy (generated by a *code producer*); and (iii) validating the binary (together with the proof) on the consumer side. The validation of a proof is quick and driven by a straightforward algorithm. Therefore, the consumer must trust only the implementation of this simple algorithm in addition to the soundness of the safety policy. The PCC approach can be considered as an extension to formal verification in establishing some properties of programs that have been settled beforehand, such as termination, lack of deadlock, or type-safety.

The work described in [Colby et al. 2000] investigates the problem of preserving properties provided by the design of a high-level programming language after being compiled into assembly language. In particular, programs in Java are type safe, but after compilation into native machine code there is no guarantee that type-safety properties are still maintained by the final code. The paper defines the safety policy corresponding to type safety for a subset of the Java language and

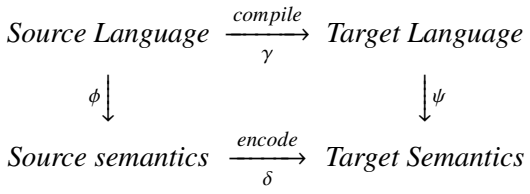

Figure 1.1: Morris correctness diagram

shows how a compiler can produce a proof that the compiled programs comply with the type-safety properties.

Since only particular properties of a compiled program are considered, the formal specification of the target languages can be very specific – in order to express and prove the required properties. In the same manner as translation validation, proof-carrying code can exploit the proof-checking approach, when only the correctness of a small and simple proof-checker remains crucial to the correctness of the overall process of establishing properties of a compiled program.

1.1.3 Algebraic compilation

The seminal work regarding *algebraic correct compilation* was proposed by Burstall and Landin [1969]. Their work presents an algebraic framework in order to prove the correctness of an expression compiler. The goal is achieved by means of successive transformations (i.e., homomorphisms) between algebras modelling different representations of the expression being compiled.

In general, traditional approaches to compilation are based in the idea of correctness expressed as the commutativity of a diagram[1] like the one presented in Figure 1.1, where the nodes are algebras [Cohn 1981] and the arrows homomorphisms. This kind of diagram was firstly introduced by Burstall and Landin but its usage was reinforced by Morris [1973]. Later on, Thatcher et al. [1979] reformulated the diagram of Figure 1.1 in the framework of initial algebra semantics [Goguen et al. 1977].

An alternative approach to the algebraic verification of compilers is the one based in the idea of a *normal form*. In general, a normal form is a "highly restricted subset of a programming language" [Hoare and He 1998], usually achieved by removing many of the operators of the language and by imposing a fixed order of application of the others. The most common usage of normal forms is a means to reduce the question of semantic equivalence into, the much easier to check, syntactic equality. Nevertheless, the main interest regarding normal forms from the axiomatic reasoning point of view lays in its usefulness to prove completeness of a certain set of rules. The rules are regarded as complete if they are sufficiently comprehensive to derive a corresponding normal form from any given program, a task that is much simpler than to prove completeness in the traditional logical sense.

The verification approach based in the idea of normal form was initially proposed by Hoare [1990] and captures the compilation process by the predicate $C\ p\ s\ f\ m\ \Psi$. It states that the code stored in m with start address s and finish address f is a correct translation of the source program

[1] The *commutativity of the diagram* is defined as the equivalence between $(\psi \circ \gamma)$ and $(\delta \circ \phi)$.

p, where Ψ is a symbol table mapping the global variables of p to their corresponding addresses in the machine's memory space. C, on the other hand, is defined by

$$C\ p\ s\ f\ m\ \Psi\ =_{df}\ \overset{*}{\Psi} \sqsubseteq \mathcal{I}\ s\ f\ m$$

where $\sqsubseteq$ is a preorder relation that captures the notion of improvement (or *refinement*), Ψ and $\overset{*}{\Psi}$ are a pair of simulation functions establishing the link between the specification and the concrete data and control spaces; and $\mathcal{I}$ is an interpreter for the target language.

The compilation C is specified by a set of theorems (algebraic rules), one for each programming construct in the language that progressively transform source-language constructs into target-language ones. The theorems are proven to preserve the semantics of the constructs during the transformation and, as they have the form of Horn Clauses [Demoen 2005], the transformation can be mechanized through logic programming [Bowen 1993].

1.1.3.1 Compilation of imperative constructs

The first application of the algebraic approach to sequential programming was proposed by Sampaio [1993; 1997] for a generic imperative programming language including procedures (supporting parameters), variable scoping, recursion and iteration. The approach is based in the idea of reduction to a normal form but it adds several relevant features:

- *Abstraction*. The approach is abstract in two ways. Firstly, the source language contains most of the features common to imperative sequential programming languages like sequential composition, iteration, variable scoping, procedural parameters, conditionals and assertions. In this way, the approach is general enough to easily cover the most common features of any imperative language. Secondly, the normal form chosen for the target language is an abstraction of a machine's architecture: it contains a program-counter-based execution cycle and an addressable memory space. This allows the compiler to not to be bound to particular architectures and makes the compilation strategy applicable to a wide range of hardware platforms.

- *Uniformity*. The whole compilation process is formalized within a single semantic framework. In particular, the author extended the generic (source) imperative language with a specification language and a *reasoning language* (axiomatic reduction rules). As this extended language is able to capture all the stages in the compilation (including the compilation rules), the preservation of the semantics is ensured by construction: all the stages in the compilation share the same semantic framework, and the reduction rules are proven to be semantics-preserving.

- *Machine support (mechanization)*. The algebraic nature of the approach allowed its formalisation using the OBJ3 term rewriting system [Goguen and Winkler 1988], providing a practical test bed for the practical applicability of the ideas presented.

As another application of the same approach, Barros and Sampaio [1994] presented some ideas towards a provably correct hardware/software partitioning. The proposal used Occam as both the source and target languages, where the algebraic transformations were justified using the laws of Occam programming [Roscoe and Hoare 1988].

In later work, Silva et al. [1997*a*;*b*; 1999; 2004] extended the original formulation and presented a five-phase compilation approach for hardware/software co-design. The most interesting feature of this idea is the inclusion of heuristics in a formal environment that pursues correctness. This is possible because of the orthogonality that the framework provides between the "decision" stages (the ones that involve heuristics), and the transformational phases. This orthogonality also allows the exchange of heuristics (to optimize the outcome according to a different criterion) without needing to revalidate the correctness of the compiler.

In [Hoare et al. 2000], a proof of correctness is provided for a version of Dijkstra's language [Dijkstra 1976] (including sequential composition, recursion and variable declarations and scoping), based on its operational semantics. The authors capture the operational semantics of the language in a normal form shaped to mimic a sequential interpreter as a loop that executes one step of the operational semantics at a time until there are no more actions to perform. Each possible step is an assignment that updates the data control state of the program. The authors also provide a proof of the fact that the interpretation of a program (i.e., its normal form) has the same meaning as the program itself.

Finally, an algebraic compiler for the object oriented paradigm of programming has been proposed by Duran et al. [2001; 2003*a*;*b*]. The proposed source language is a subset of the Java language (including, among other features, inheritance, visibility rules and dynamic creation of objects). The definition of methods was also covered, with a sub language specially tailored for this purpose (based in Morgan's language [Morgan 1990]). In order to be able to cope with the dynamic creation of objects, the original normal form proposed by Sampaio [1993] was extended with a stack and explicit stack-handling primitives.

Thanks to the algebraic nature of the approach, the translation mechanism has been automated [Oliveira et al. 2002] and the soundness of the rules used in the derivation proven correct [Borba et al. 2004] with respect to weakest precondition semantics for the language [Cavalcanti and Naumann 2000].

1.1.3.2 Hardware compilation

Work has also been undertaken by He and others to transform programs written in a subset of Occam into a normal form suitable for an implementation in hardware [He et al. 1993; Bowen et al. 1994]. In this work, a circuit is represented as a program with the particularity of having the state of the "program" formed by the control and data paths of the generated hardware.

The translation is completed by defining a normal form comprised of an assumption about the activation of the generated circuits; an execution loop; and an assertion that determines the final state of the computation. A simplification of timed processes is also used to model the state change of both the control and the data path of the circuits. With the algebraic rules provided in this work, the authors show how an arbitrary source program can be reduced to this special normal form.

The translation from the normal form into net-lists (a special list of gates and latches that is one of the most common forms of hardware description) is regarded as "trivial" and only addressed informally.

Another approach towards hardware synthesis based on the same principle was proposed by Iyoda and He [2001*b*]; He [2002]. In this work, the authors describe an algebraic approach towards the compilation of Verilog programs into hardware descriptions. The approach has also been mechanized in Prolog [Iyoda and He 2001*a*] and has been extended to cope with hardware/software partitioning [Qin et al. 2002]. The main weakness of this work is the much reduced subset of Verilog that it addresses (absence of loops or recursive constructs). As a solution to the lack of treatment of iterative constructs, the authors proposed the integration of their work with a hardware compilation scheme proposed by Bowen and He [2001]. Unfortunately, no follow-up of this idea has been found in the literature to the extent of our knowledge.

1.2 Our approach to compilation

The approach to compilation presented in this work follows the normal-form based algebraic compilation strategy described in the previous section. The main reason for our choice lies in the limited success of approaches based in *a posteriori* formal verification and the inability of PCC to provide full coverage of all aspects regarding the correctness of a compiler (mainly due to the fact that PCC was not devised as a verification technique). Furthermore, we have selected the algebraic approach over other successful cases of correctness by construction using formal methods mentioned in Section 1.1.1, because we are pursuing a framework where our proofs can be conducted purely by algebraic means and where we can reason about non-terminating programs. Finally, the algebraic technique has been successfully applied to the hardware compilation of Occam [Barrett 1992] and subsets of Verilog [Thomas and Moorby 1998].

In the context of our compiler, all normal forms are state-machine representations of the source program. Each of our state-machine-based normal forms is an iteration over a set of steps describing how the machine progresses throughout its execution. In turn, steps represent the combinatorial and variable updates that can be carried out within a single clock cycle. The normal form follows the source program's control flow by means of guarding the execution of each step by a condition capturing the specific control state in which the step should be activated.

In the process of gradually transforming the source code into its compiled form, we want to allow the possibility of any of the intermediate normal forms to introduce improvements to the program. This notion of a program being better than another one is known as *refinement* and it is formally defined as a relationship between programs denoted by the symbol $\sqsubseteq$. $P \sqsubseteq Q$ stands for any possible behaviour of Q being a possible behaviour of P. Our compilation process can then be formally stated as the following sequence of transformations:

$$\text{source program} \underset{\substack{\textit{compilation}\\ \textit{theorems}}}{\sqsubseteq} \text{first normal form} \underset{\substack{\textit{compilation}\\ \textit{theorems}}}{\sqsubseteq} \cdots \underset{\substack{\textit{compilation}\\ \textit{theorems}}}{\sqsubseteq} \text{final normal form}$$

By following the normal-form based compilation strategy, our approach shares the aforementioned benefits of this kind of the algebraic techniques: abstraction, uniformity and mechanisation. There are, however, shortcomings in the general usage of the algebraic approach to compiler design. Sampaio [1997] identifies four main limitations to this approach:

1. *The difficulty in the implementation of debuggers for the compiled program*. This is a consequence of the difference in the level of abstraction that is introduced by the reduction theorems leading to refinement. In turn, this makes the traceability of the object program back to source code very complicated, if not impossible.

   On the other hand, the main concern of this work is correctness in a context where the source program will be developed within rigorous development environments. In this context, the compiled program acts as the implementation and the source program as the specification. As the compiler is known to be correct, any errors in the compiled code must be contained within the original source program, where other techniques can be used to identify and eliminate them.

   Finally, if debugging is an unavoidable need, it is still possible to use commercial simulators and compilers for the language to debug the source program until the desired level of confidence is reached. Once the source program is ready for its production, our correct compilation approach can be applied to produce the release form of the compiled program.

2. *The lack of existing algebraic treatment of optimisations*. In our context, some simplifications are carried out along with the compilation process. Nevertheless, in order to be able to compete with the optimised output of existing commercial tools, an optimisation stage should be added before the actual generation of hardware. We regard this as a future extension of our work.

3. *The possible differences arising from the final normal form representation and the actual target machine*. This limitation arose in the context of the compilation of imperative programs due to the fact their normal forms are abstractions of an interpreter executing the program instructions. In this context there is a possibility that there will be a semantic gap between the interpreter-based abstractions, and the actual machines executing the code produced by the compiler. In our work, the synthesised components implementing the source program are comprised by very basic, well-understood hardware components[2]. As the normal form is directly mapped to these components with very clear semantics, we believe this limitation does not apply to our work.

4. *The correctness of the basic laws in the reasoning language*. The basic laws in the reasoning language are postulated as axioms expressing the relationship between the operators of the language. The method of postulating axioms is questioned as incorporating an inconsistent

[2]There has been a great amount of research in this field in the last years. For an overview of the available techniques and results, the reader is referred to [Kropf 1997; Srivas et al. 1997; Perry and Foster 2005].

law would allow one to prove any result, including the correctness of an inaccurate compiler. We address this problem by providing a denotational semantics for the input language and its reasoning extensions in the context of the Unifying Theories of Programming (UTP). Our UTP denotational semantics not only provides a model for our theory but also allows us to prove all the axioms in the compilation framework.

The following sections present an overview of our source and reasoning languages, as well as the basic principles by which we can embed the former within the latter. We also explore the main definitions and most relevant features of the normal forms used in our compiler, and illustrate the general aspects of the technique with an example.

1.2.1 Our input language

We have decided to adopt Handel-C [Celoxica Ltd. 2002*a*] as the input language for our compiler. Handel-C is a Hardware Description Language (HDL) based on the syntax of the ANSI C [ISO 1999] language extended with constructs to deal with parallel behaviour and process communications based on CSP [Hoare 1983]. The language is designed to target synchronous hardware components with multiple clock domains, usually implemented in Field Programmable Gate Arrays (FPGAs).

Our preference of Handel-C over other more popular HDL languages (such as VHDL [IEEE 1993], Verilog or Occam) is based in the following reasons: (i) Handel-C features the combination of shared variables in a context of parallel processes and communications that has not been addressed in any verified compiler; (ii) it contains most of the features present in state-of-the-art HDLs, allowing our work to be general enough to be extended to other languages in this group; and (iii) being a "high-level programming language with hardware output" [Kamat et al. 2009] it is suitable for reasoning within the algebra of programs without the need of introducing too many hardware-level concepts.

For this work, we have adopted the simplified subset of Handel-C presented in Figure 1.2. Most constructs in the language can be built by combining constructs in our subset, with exception of function calls, pointers and hardware-optimisation features. There is no dynamic allocation of memory in Handel-C so the usage of pointers is as a mean of referring to statically created data objects or functions. In this sense, we believe the treatment of pointers will not be a significant contribution to contents of this thesis. On the other hand, functions and hardware optimisations are means to achieve a better utilisation of the available space and features within the FPGA. We regard the treatment of these constructs as future extensions, more details on our ideas on how to incorporate them to this work are presented in Chapter 7.

As described in the language documentation by Celoxica Ltd. [2002*a*], programs are comprised of at least one **main** function and, possibly, some additional auxiliary functions. Multiple main functions within the same file produce the parallel execution of their bodies. It is possible to produce the same effect in our reduced subset by means of the parallel operator.

All C-based constructs in Handel-C behave as defined in ANSI-C [Kernighan and Ritchie 1988] but with some additional restrictions regarding the clock-based, synchronous nature of the

⟨program⟩ ::= **main(){** ⟨statements⟩ **}**

⟨statements⟩ ::= ⟨statement⟩ $\fatsemi$ ⟨statements⟩
| ⟨statements⟩ $\|_{HC}$ ⟨statements⟩
| **if** ⟨boolean expression⟩ **then** ⟨statements⟩ **else** ⟨statements⟩
| **while** ⟨boolean expression⟩ **do** ⟨statements⟩
| ⟨statement⟩

⟨statement⟩ ::= ⟨variable list⟩ $:=_{HC}$ ⟨expression list⟩
| **delay**
| ⟨comm-guard⟩
| **priAlt** {⟨case-guards⟩}

⟨case-guards⟩ ::= **case** ⟨comm-guard⟩**:** ⟨statements⟩ $\fatsemi$ **break** $\fatsemi$ ⟨case-guards⟩
| **case** ⟨comm-guard⟩**:** ⟨statements⟩ $\fatsemi$ **break**
| **default:** ⟨statements⟩

⟨comm-guard⟩ ::= ⟨channel name⟩?⟨variable name⟩
| ⟨channel name⟩!⟨expression⟩

Figure 1.2: Restricted syntax for Handel-C programs

language. The evaluation of expressions is performed by means of combinatorial circuitry, and it is completed within the clock cycle in which they are initiated thus expressions are considered to be evaluated "for free" [Page and Luk 1991] due to this semantic interpretation. This way of evaluating conditions affects the behaviour of all constructs in the language regarding the time they take to complete. In the case of selection, the branch selected for execution will start execution within the same clock cycle in which the whole construct is initiated. The **while** construct starts its body in the same clock cycle in which the looping condition is evaluated. The whole construct terminates within the same clock cycle in which its condition becomes false. Assignment, on the other hand, happens at the end of the clock cycle. The expressions that denote the values to be used to update variables must be side-effect free (this is a Handel-C restriction). Furthermore, as the combinatorial gates used to implement functions always produce a result, we assume all expressions are total and terminating[3].

From the remaining non-C constructs, parallel composition of statements executes in a *real* parallel fashion as it refers to independent pieces of hardware running in the same clock domain. **delay** leaves the state unchanged, but takes a whole clock cycle to finish. Input and output have the standard blocking semantics [Hoare 1983]: if the two parts are ready to communicate, the

[3] In case the treatment of partial expressions is necessary, our theory can easily be extended to incorporate this notion by means of using the technique introduced by Hoare and He [1998, Page 78].

value outputted at one end is assigned to the variable associated with the input side. Both sides of the communication take one full clock cycle to successfully communicate. A process trying to communicate over a channel without the other side being ready will block (delay) for a single clock cycle and attempt the communication again in the next clock cycle until it is able to synchronise. Finally, the **priAlt** construct attempts each of its alternatives in order until: (a) one of them holds true and the corresponding actions are activated; (b) it reaches a **default** guard, in which case it unconditionally initiates the sub-program associated with it; or (c) it reaches the end of the guard list without being able to activate any guard, which implies the fact that there was no **default** clause. In the case of (c) the **priAlt** construct delays for a whole clock cycle and attempts the same strategy again in the next clock cycle.

1.2.2 The reasoning language

The reasoning language is comprised by three sub-languages integrated under the same semantic framework: (a) the source language; (b) the laws and constructs intended for reasoning about programs; and (c) the normal forms, including the one capturing our target language. Figure 1.3 shows the aspects of our reasoning language that capture the main programming constructs in the context of shared variables and parallelism. In our description of the language operators, we use x to stand for an arbitrary program variable, b for a boolean expression and P and Q for programs.

$I\!I_1$	one clock-cycle delay
$(x \overset{:=}{_{\text{snc}}} e)_1$	one clock-cycle assignment
$P; Q$	sequential composition
$P \parallel_{\hat{M}} Q$	shared-variables parallel composition
$P \lhd b \rhd Q$	selection: **if** b **then** P **else** Q
$b * P$	iteration: **while** b **do** P

Figure 1.3: Programming constructs in the reasoning language

Even though the precise semantics of these constructs will be given in Chapter 3, we present here an informal description of their meaning:

- $I\!I_1$ is the program that keeps the variables constant, and takes one clock-cycle to terminate.
- $(x \overset{:=}{_{\text{snc}}} e)_1$ is a one clock-cycle update of x with the value of expression e.
- $P; Q$ stands for the execution of P followed by Q.
- $P \parallel_{\hat{M}} Q$ describes the parallel execution of P and Q. Both, P and Q will receive private copies of their shared variables. At the end of every clock cycle, the local copies will be merged by means of the predicate $\hat{M}$ and the resulting value will be passed on to P and Q as the initial value of their private copies for the next clock cycle.
- $P \lhd b \rhd Q$ stands for the selection of P or Q depending on the value of condition b.

- $b * P$ executes the program P while condition b holds. When b does not hold, the whole construct terminates immediately leaving the state unchanged.

There is a clear correspondence between a major part of the restricted subset of Handel-C we use as the input of our compiler and the programming constructs within the reasoning language. This relationship is formalised by means of the denotational semantics for Handel-C we introduce in Chapter 4. In this way, we formally establish the link between the source language and programming constructs defined above, effectively producing an embedding [Boulton et al. 1993] of Handel-C in the reasoning language. We take advantage of this link, together with the fact that we also have precise UTP semantic definitions for the reasoning language, to achieve one of the contributions of this work: the formal proof of the basic compilation axioms.

The second aspect of our reasoning language comprises the specification constructs and the algebraic laws used to characterise these operators. Figure 1.4 presents a comprehensive list of the operators in the specification space of the reasoning language. Again, let x stand for an arbitrary program variable, b for a boolean condition, ch for a channel; and a, P and Q for programs (or specifications). An informal account of these operators is given below, yet their definitions and

Π	the program with no effect
$x \overset{:=}{_{\text{snc}}} e$	assignment
$P \parallel Q$	disjoint parallel composition
`var` x	open scope for variable x
`end` x	close scope for variable x
$\top$	miracle
$\bot$	abort
$b^{\top}_{\text{S}}$	assumption ($\Pi \lhd b \rhd \top$)
$b^{\bot}_{\text{S}}$	assertion ($\Pi \lhd b \rhd \bot$)
$b \underset{\text{snc}}{\longrightarrow} P$	guarded command ($P \lhd b \rhd \mathit{II}_1$)
$b * P \blacktriangleright Q$	iterated selection ($(b * P); Q$)
in-req(ch)	input request over channel ch
out-req(ch)	output request over channel ch
rd(ch)	checks whether there has been an input request on ch or not
wr(ch)	checks whether there has been an output request on ch or not
in(ch)	function returning the value being transmitted over ch
out(ch, e)	sends the value e over ch
case $a; b \,?\, P \mid Q$	case construct ($a; (P \lhd b \rhd Q)$)

Figure 1.4: The specification space

algebraic laws are presented in detail in Chapters 3 and 4.

- Π is the program that finishes immediately and has no effect over the environment.

- $x \overset{:=}{_{\text{snc}}} e$ is an *immediate* assignment of the value e to variable x. In the context of hardware, we regard this assignment as the transferral of value e into wire x. Note that x cannot be a flip-flop because these kind of devices take a whole clock cycle to perform an update.

- $P \parallel Q$ stands for the parallel execution of P and Q when they do not have variables in common.

- The pair `var` x and `end` x are used as dynamic scope delimiters for a given variable x.

- Miracle ($\top$) stands for the program with the most defined behaviour that can satisfy any specification. This is a very useful theoretical concept when reasoning about programs but it is, clearly, infeasible since it cannot be implemented.

- Abort ($\bot$) is the opposite extreme to miracle and it has the most undefined behaviour possible: it may terminate with an arbitrary result or it may even not terminate. Following Hoare [1983] we identify abort with all programs that might diverge before performing any action visible to its environment.

- The program $b_{\text{S}}^{\top}$ models an assumption: it can be ignored if b holds (i.e., it reduces to $\mathbb{I}$), and it behaves like miracle otherwise. An important consequence of its behaviour when b does not hold is that it frees the implementation from any commitment as miracle ($\top$) will satisfy any requirement for it.

- Similarly, $b_{\text{S}}^{\bot}$ models an assertion that behaves like $\mathbb{I}$ if b holds or like abort ($\bot$) otherwise.

- $b \overset{\rightarrow}{_{\text{snc}}} P$ is the synchronous version of Dijkstra's guarded command [Dijkstra 1976]: it behaves like P if b holds, it reduces to II_1 otherwise. This behaviour in the case where b does not hold can be explained in the context of hardware components, where even though P does not get activated, the program variables need to be preserved and, as mentioned before, establishing a flip-flop's value (even if it is the same variable held there already) takes a whole clock cycle to be completed.

- $b * P \blacktriangleright Q$ is a particular form of the iteration construct presented in Figure 1.3: it operates like the standard iteration $b * P$ while b holds, then it transfers the control Q. Due to the way in which we encode information in the normal forms for the compilation, our compiler only accepts programs with loops that can be converted into this form. For further details see Section 4.2.2.

- The actions **in-req**(ch) and **out-req**(ch) model the program's readiness to perform an input/output over channel ch at the current clock cycle.

- The condition **rd**(ch) / **wr**(ch) characterises whether there is a reader/writer (i.e., a process performing an input/output) operating on channel ch at the current clock cycle.

- The action **in**(ch) / **out**(ch, e) implements a non-blocking unbuffered communication protocol over channel ch. **in**(ch) returns the value being transmitted over channel ch during the

current clock cycle without verifying if any process has actually performed an output to it. Conversely, **out**(ch, e) will force the transmission of e over channel ch even if there is no process ready to receive it.

- **case** $a; b \ ? \ P \mid Q$ is a particular form of the case construct that performs the actions in a before evaluating b. Control is transferred to P if b holds or to Q otherwise.

The final aspect of the reasoning language comprises the normal forms used along the compilation process. Our compiler's first normal form eliminates the C-based structure of the program by means of expressing it in a state-machine-based representation. The most relevant aspects of this representation are:

- *Encoding of the control flow within the state of the machine*. In practical terms this implies the introduction of state variables and the association of unique values to each possible control-flow state.
- *Each construct is encoded as a set of one clock-cycle steps*. Each step is guarded by a control condition capturing the exact control state in which the action should be activated.
- *Steps in the normal form are combined by means of the* $\|_{\hat{M}}$ *operator described before*. Several steps can be active at any given clock cycle, making the state machine capable of parallel behaviour.

Along the thesis, we abbreviate the first normal form and write

$$a : [s, \left(b_1 \underset{\text{snc}}{\rightarrow} P_1 \parallel_{\hat{M}} b_2 \underset{\text{snc}}{\rightarrow} P_2\right), f]$$

where:

- a is a list of control variables governing the execution of the normal form;
- s is a condition describing the initial control state of the machine;
- $(b_1 \underset{\text{snc}}{\rightarrow} P_1 \parallel_{\hat{M}} b_2 \underset{\text{snc}}{\rightarrow} P_2)$ is the set of steps the machine can perform. b_1 and b_2 are conditions guarding the activation of each of the steps, as with s, they are control-based conditions. This set of steps will be iterated until no step can be activated (denoted by the condition $b_1 \vee b_2$).
- f is a condition reflecting the final control state the machine should be in when it terminates.

As mentioned before, the normal forms can be expressed in terms of operators from the reasoning language. The above semantic description of the first normal form can be expressed as:

$$\texttt{var}\ a; s_{\mathsf{S}}^{\top}; (b_1 \vee b_2) * \left(b_1 \underset{\text{snc}}{\rightarrow} P_1 \parallel_{\hat{M}} b_2 \underset{\text{snc}}{\rightarrow} P_2\right); f_{\mathsf{S}}^{\perp}; \texttt{end}\ a$$

In turn, our second (and final) normal form keeps the underlying state-machine representation, yet aims at simplifying the elements used in the normal form so they can be directly mapped into

well-understood hardware components. The most relevant transformations carried out from the first- to the second-normal form are as follows:

- *Isolation of expressions used in the right-hand side of assignments for their implementation in combinatorial logic*.
- *Introduction of wires to interconnect and transfer values between the different operational units being generated in the FPGA*.
- *Simplification of the parallel by merge combination of steps into disjoint-alphabet parallel assignments*.

Perhaps the most interesting consequence of the simplification of the parallelism carried out to reach our second and final normal form is that it only contains a single step of the form:

$$w \overset{:=}{_{\text{snc}}} f_w(v); (v \overset{:=}{_{\text{snc}}} f_v(w))_1$$

where:

- $f_w(v)$ is a function updating the wires w based on the program and control variables in v.
- $f_v(w)$ is a function selecting which of the values carried by w should be used to update the program and control variables in v.

The actions on the left of the sequential composition are meant to be implemented in combinatorial logic, while the update of the program and state variables on the right modifies registers and will take one clock cycle to be completed. In the same way we have abbreviated the first normal form, the notation we introduce to denote our second normal form is as follows:

$$a, w : [s, b * (P), f]$$

where:

- a, s and f have the same interpretation they had in the first normal form;
- w is the list of wires used in the normal form to carry values among the different components when mapped into hardware, they are implicitly assumed to initially hold the value **false**.
- b is the looping condition of the normal form. It is the same condition used to control the execution loop in the normal form ($b_1 \vee b_2$ in our first normal form above);
- P is a second normal form step as described above;

As with our first normal form, the behaviour of the second normal form can be described in terms of operators from the reasoning language:

$$\texttt{var}\ a, w; (s)^{\top}_{\mathrm{S}}; b * P; (f)^{\bot}_{\mathrm{S}}; \texttt{end}\ a, w$$

1.2.3 The approach through a simple example

In this section we give an overview of the compilation approach outlined above based on a single assignment:

$$x \overset{:=}{_{\mathsf{HC}}} e$$

This program should assign the value of the expression e to x and terminate after one clock cycle. Following our compilation strategy, this behaviour can be implemented by means of the following normal form:

$$a : [a = s, (a = s \rightarrow (x, a \overset{:=}{_{\mathsf{snc}}} e, f)_1), a = f]$$

Expanding our definition of normal form, we obtain the underlying state machine representing our simple program:

$$\begin{array}{l} \texttt{var}\ a; \\ \quad (a = s)^{\top}_{\mathsf{S}}; \\ \quad * ((a = s) \rightarrow (x, a \overset{:=}{_{\mathsf{snc}}} e, f)_1); \\ \quad (a = f)^{\bot}_{\mathsf{S}}; \\ \texttt{end}\ a \end{array}$$

This machine can be further refined to introduce wires to carry the value produced by expression e (calculated by combinatorial logic) into the memory-capable devices holding x (e.g., an FPGA implementation of a flip-flop). A similar strategy can be applied to the control variable a to obtain the following second-normal-form representation of our program:

$$a, w_1, w_2 : [a = s, (a = s) * \begin{pmatrix} w_1 \overset{:=}{_{\mathsf{snc}}} (e \lhd a = s \rhd x) \parallel w_2 \overset{:=}{_{\mathsf{snc}}} (f \lhd a = s \rhd a); \\ (x, a \overset{:=}{_{\mathsf{snc}}} w_1, w_2)_1 \end{pmatrix}, a = f]$$

From our definition of the second normal form we can present the state machine corresponding to the original program:

$$\begin{array}{l} \texttt{var}\ w_1, w_2, a; \\ \quad (a = s)^{\top}_{\mathsf{S}}; \\ \qquad (a = s) * \begin{pmatrix} w_1 \overset{:=}{_{\mathsf{snc}}} (e \lhd a = s \rhd x) \parallel w_2 \overset{:=}{_{\mathsf{snc}}} (f \lhd a = s \rhd a); \\ (x, a \overset{:=}{_{\mathsf{snc}}} w_1, w_2)_1 \end{pmatrix}; \\ \quad (a = f)^{\bot}_{\mathsf{S}}; \\ \texttt{end}\ w_1, w_2, a \end{array}$$

1.2.4 The semantics of the reasoning language

We have already mentioned that the correctness results from the algebraic compilation technique may be invalidated by the introduction of an incorrect axiom in the set of algebraic laws the compiler is based on. In this thesis we address this possible problem by means of providing a denotational semantics model for the reasoning language that we use to prove the validity and consistency of the axioms used in the compiler.

The goal is to provide a semantic framework in which it is possible to reason about synchronous, parallel programs with shared-variables. As there are several existing hardware description languages with most or all the features required, our initial intention was to try to apply the same mathematical domain used in their semantics as the semantic framework for our work. Perhaps the most widely used hardware description language is VHDL [Ashenden 1999]. There are several operational [Goossens 1995; Nicola and Hennessy 1984; De Nicola and Pugliese 1994], denotational [Breuer et al. 1994*b*;*a*] and logical [Breuer et al. 1995] semantic models for it. The standard defining VHDL [IEEE 1993] informally describes the meaning of all the language's constructs by means of the effect they have over an interpreter (i.e., the standard describes the simulation semantics for the language). Unfortunately, all the semantic models mentioned above follow this interpreter-based approach, making all of them unsuitable for proving the kind of algebraic laws needed for our compiler. Furthermore, VHDL is a language meant to give the designer a great deal of control over the hardware being generated. From the language perspective, this means the constructs tend to reflect the hardware structure rather than the kind of high-level programming features we are interested in our input and reasoning languages (e.g. selection, assignment, sequential composition, iteration, etc.).

Occam [Barrett 1992] is a higher-level language that can be used to generate hardware. Denotational semantics based on the failures-divergences model [Roscoe 1997] have been proposed for the language [Roscoe 1985; Roscoe et al. 1993]. In addition to these, an axiomatic characterisation of the language was proposed in [Roscoe and Hoare 1988], together with a proof of equivalence between this and its denotational semantics. Even though this semantic approach seems to be more in line with our needs, there are two main drawbacks if we were to use this denotational model as the foundation for our reasoning language: (a) the lack of explicit representation of time; and (b) the fact that variables cannot be shared among parallel processes. Furthermore, a direct consequence of observation (b) is that there is no need for synchronisation as a mechanism to keep variables consistent among parallel processes.

Handel-C, on the other hand, presents all the features we are interested in, and its underlying synchronicity forces any semantic model for the language to account for the clock-cycle-based timing structure of all constructs. Several operational [Butterfield 2001; Butterfield and Woodcock 2002; 2005*b*] and denotational [Butterfield and Woodcock 2005*a*; Butterfield 2007; Butterfield et al. 2007; Perna and Woodcock 2007] semantic models have been proposed for Handel-C. In all of these works, the semantics are represented in terms of sequences indexed by clock-cycle where each element in the sequence is a collection of sub-atomic events describing the steps the hardware carries out to compute and store results. Unfortunately, this "operational" nature of the semantics

incorporates too many implementation details, hindering the possibility of performing any of the proofs about the reasoning language we intend to conduct.

Given the fact that the existing semantics for HDLs did not provide us with the kind of semantic domain we need for our work, we considered two additional formalisms as possible semantic domains: Higher-Order Logic (HOL) [Gordon and Melham 1993] and the weakest precondition (wp) calculus [Dijkstra 1975; 1976]. In the case of HOL, it has been extensively used in the specification, generation and verification of hardware components [Melham 1993; Iyoda 2007; Berghofer and Strecker 2003; Klein and Nipkow 2003], and in the verification of HDLs themselves [Boulton et al. 1993; Perna and Woodcock 2008]. However, HOL only offers very basic direct support for programming constructs [Norrish and Slind 2007, Chapter 2] like the ones required in our context and this will force us to define a whole theory of programming within HOL. This is not a good choice for us as there are other alternatives where there is existing support for most of the programming constructs we want to include in our framework.

On the other hand, our reason for considering the wp-calculus lies in the fact that it was used as the semantic foundation for an algebraic compiler for an object-oriented language [Cavalcanti and Naumann 2000; Duran et al. 2001]. The main limitation we observe over the wp-calculus as the semantic framework for our work is its poor integration with other theories within the same semantic domain. For example, one of the possible extensions of our work is to include pointers in our input language. In these regards, we would like to have a way of linking our semantics with a theory of pointers rather than having to re-formulate the entire semantic framework.

In the light of the observations and weaknesses with all the approaches mentioned above, we selected the Unifying Theories of Programming (UTP) [Hoare and He 1998] as the semantic domain for our reasoning language. Our choice for the UTP is mainly based in two factors. Firstly, the UTP provides definitions for all the operators needed in our work. In this way, theories in the UTP do not need to re-define them: they only need to show the operators are closed in the theory (i.e., the result of combining constructs in the theory is itself within the theory). This means that even if we define a new theory for our context (as we do in Chapter 3), we only need to analyse how the programming operators behave under the conditions imposed by the new theory. Secondly, all programming constructs are defined as predicates and a single notion of refinement is used across all theories (i.e., reverse logical implication under universal closure over all known variables). This makes each of the UTP theories a lattice [Davey and Priestley 2002] and it is possible to establish links among them by means of Galois connections [Erne et al. 1992]. This is one of the key aspects of the UTP as it allows an easy integration and exchange of results between different theories and programming paradigms.

Within the UTP, our initial intention was to use the theory of designs (see Chapter 2) as the semantic domain for our reasoning framework since it contains all the features required in our context together with a comprehensive set of algebraic laws that are of great advantage for our reasoning language. While trying to prove additional laws needed in our context, we discovered a series of limitations that hindered the usability of the design theory as the semantic foundation for our reasoning framework. This is mainly related with assertional reasoning after clock-cycle boundaries. We solve this and other problems by restricting the theory of designs with additional

conditions that allow the kind of assertional reasoning we need for synchronous, parallel programs with shared variables. We use this new theory of synchronous designs (see Chapter 3) not only as the semantic domain to prove the laws in the reasoning language, but also to give semantics to Handel-C (described in Chapter 4) which is in line with the needs of our work.

1.3 Objective and contributions of this work

This thesis aims at producing a verified compilation approach for the Handel-C programming language. The compilation strategy translates a formal model of the source program into a formal abstraction of the hardware to be implemented in FPGA devices.

In the light of the objective above, the main contributions of this thesis are as follows:

- **A theory of synchronous designs in the UTP**. A semantic framework that allows the reasoning and proving of algebraic laws in the context of synchronous, shared-variable, parallel programs.
- **Semantic model for the reasoning language**. A shallow embedding [Boulton et al. 1993] of the compilation framework in the UTP theory of synchronous designs. From this underlying semantic model we have proved (rather than just postulated) the basic axioms of the compiler, hence addressing one of the major weaknesses of the algebraic compilation approach.
- **UTP semantics for Handel-C**. A formal semantics for the Handel-C language that we use to show a number of algebraic laws and equivalence relationships about Handel-C programs. As the reasoning language is embedded in the UTP theory of synchronous designs, our semantics for Handel-C also acts as the link to have the source language embedded in the reasoning language.
- **Verified hardware synthesis for Handel-C**. A correct-by-construction compiler from a comprehensive subset of Handel-C into net-list descriptions of hardware using the algebraic approach to compilation.

1.4 Thesis structure

This thesis is divided into three parts. *Preliminaries* are the necessary topics that must be covered to place the research in context. The second part, *Handel-C, its semantics and extensions for reasoning*, presents our semantic model for the Handel-C language in the Unifying Theories of Programming [Hoare and He 1998]. It also proves the basic properties about the reasoning language operators that will serve to prove the compilation theorems. The third part, *Reducing Handel-C to net-lists*, is concerned with the formulation of the normal forms used in the compilation, the reduction theorems that establish the link between them and the mapping of the final normal form into hardware.

Chapter 2 gives a background on the UTP and presents its two most basic theories: the alphabetised relational calculus and the theory of designs. This chapter also covers the definition of the basic operators that will be used across all theories together with the healthiness conditions associated to them. The discussion then focuses on the laws and theorems that can be proved from the different operators and that will be useful in later chapters.

Chapter 3 presents the limitations of the theory of designs as a semantic domain in the context of synchronous, shared-variables, parallel environments addressed in this work. These issues are addressed by means of introducing a new theory by deriving the appropriate healthiness conditions to be imposed over the theory of designs. The rest of this chapter explores the meaning of the basic operators in the context of our new theory and shows how the basic compilation axioms can be proved within this semantic framework

Chapter 4 begins by providing denotational semantics for our input language (i.e., the restricted subset of Handel-C described in the previous section) in the theory of synchronous designs introduced in Chapter 3. We finish the chapter by defining the additional reasoning constructs described in Figure 1.4. Most of these constructs will be used in the compilation of Handel-C constructs with complex behaviour (e.g., **priAlt**, iteration and the communication primitives).

Chapter 5 describes the normal forms we use in our compilation and provides the reduction theorems for all the sequential aspects of Handel-C together with the parallel composition construct. The chapter also illustrates the compilation strategy with a set of examples and presents the strategy to map the final normal form into hardware.

Chapter 6 addresses the compilation of the input, output and **priAlt** constructs into hardware following the same approach used for the sequential aspects of Handel-C. The chapter also illustrates the compilation of the communication primitives and the **priAlt** construct with an example.

Finally, Chapter 7 presents the conclusions and future possibilities for this work. In this chapter we also explore in more detail the published work that is most closely related to this thesis.

Part I

Preliminaries

Chapter 2

Unifying Theories of Programming

"*A computer program is identified with*
the strongest predicate describing every relevant
observation that can be made of the behaviour
of a computer executing that program"

— C.A.R. Hoare

This chapter is devoted to the formalism we intend to use as the domain for the denotational semantics for Handel-C: The Unifying Theories of Programming (UTP). The relevance of this chapter lies in the fact that it presents the foundation from which the operators of the reasoning language for hardware components can be defined, and their algebraic laws can be verified.

Section 2.1 presents a broad outline of the UTP and the way in which the different theories are related to each other. In Section 2.2 we explore the most basic idea in the UTP: the alphabetised relational calculus. We also define basic programming operations, such as sequential composition and conditional that all the other subsequent theories will use. We conclude this section by briefly describing the main limitation of the alphabetised relational calculus: its inability to deal with diverging behaviour.

To solve the issues in the alphabetised relational calculus, further restrictions and additional observations are incorporated to define the design sub-theory, as described in Section 2.3. In this section we also define operators to combine designs, such as different forms of parallel composition and also show how the definitions from the previous section are closed under the design theory. We also explore filtering predicates (known as *healthiness conditions*) associated to the design theory and the additional laws that can be proved in this more restricted context.

The UTP encompasses theories other than these mentioned above, such as the reactive processes theory used to give semantics to formalisms like CSP [Hoare 1983] and higher-order/declarative programming that is used to unify functional languages such as Haskell [Hudak et al. 1992]. Nevertheless, for the purpose of this thesis we intend to introduce only those concepts from the UTP that are needed for the compilation of Handel-C programs. The interested reader is referred to [Hoare and He 1998; Woodcock and Cavalcanti 2004; Cavalcanti and Woodcock 2006] for further details about additional theories within the UTP.

2.1 An overview of the Unifying Theories of Programming

The Unifying Theories of Programming are the result of a research effort towards finding a unified framework to explain and relate different programming paradigms, from imperative and sequential, to functional and parallel.

The key idea of the UTP is to use the same underlying mathematical model to give semantics to the main features of each programming paradigm. In this way, different languages and paradigms can be reduced to this common semantic framework, and be compared with each other. Programs are interpreted as relations between an initial and a subsequent (intermediate or final) observation of the behaviour of program execution.

In the UTP, each programming paradigm is associated with a theory. Each theory can, in turn, be identified by its three main elements: an *alphabet*, a *signature*, and a set of *healthiness conditions*.

A theory's alphabet is a set of variable names that provides the vocabulary for the theory in question. It identifies observational variables whose values are relevant to characterise system behaviours. The initial observations of each of these variables are undecorated and compose the input alphabet ($in\ \alpha$) of a relation. Subsequent observations are decorated with a dash and compose the output alphabet ($out\ \alpha$) of a relation. The alphabet of each theory also contains special variables relevant to the description of its programs. For example, in the theory of designs [Hoare and He 1998, Chapter 3], the boolean variable ok captures whether the program has started or not. The observation ok', on the other hand, has the same interpretation but with respect to the program's termination.

On the other hand, a theory's signature describes the syntax denoting the objects of the theory. The meaning of every specification is given as a predicate that is restricted to the selected alphabet and signature.

Finally, the healthiness conditions precisely characterise the objects of interest in the theory from the whole set of predicates expressible by the syntax. Healthiness conditions are predicates restricting which programs belong to the theory of interest and they are useful in the unification of theories, differentiation of paradigms into families, clarification of choices in a programming language design, and so forth.

The following two sections provide more details about the two theories that serve as the semantic foundation for our work: the alphabetised relational calculus and the theory of designs.

2.2 Alphabetised relational calculus

This section briefly describes the most basic approach used to express programs as boolean predicates within a suitable alphabet. The material we present here is meant to be introductory and only covers the features necessary in later sections. For a complete account of the contents in this section we refer the interested reader to [Hoare and He 1998, Chapter 2].

The alphabetised relational calculus is the most basic UTP theory and provides the definitions for most of the programming operators that will be used by all subsequent (and subset) theories.

As indicated by its name, programming constructs in this theory are formalised as an alphabet-predicate pair, where the predicate's free variables are all members of the alphabet. Furthermore, the predicates in the theory establish the relationship between undecorated variables (i.e., the initial state) and their decorated counterparts (i.e., the final state). The standard predicate calculus operators can be used to combine alphabetised predicates. Their definitions, however, have to specify the alphabet of the combined predicate.

The alphabetised relational calculus provides definitions for most of the imperative-programming constructs (e.g., selection, sequential composition, assignment, etc.). The selection construct provides the possibility of choosing one of two possible programs for execution in accordance with the truth value of a condition. In the alphabetised relational calculus, this need is addressed by means of an infix syntax version of the conditional operator. Informally, $P \lhd b \rhd Q$ stands for P if b else Q. More formally, the definition of the conditional operator is as follows.

Definition 2.2.1. *Conditional*

$$P \lhd b \rhd Q \ =_{df} \ (b \wedge P) \vee (\neg b \wedge Q)$$

provided $\alpha b \subseteq \alpha P = \alpha Q$

We will sometimes want to use the conditional construct within expressions, making P and Q expressions themselves (rather than predicates as in the definition above). When used as a expression, our definition for the conditional construct is as follows:

Definition 2.2.2. *Conditional operator for expressions*

$$e_1 \lhd \mathbf{true} \rhd e_2 \ =_{df} \ e_1$$
$$e_1 \lhd \mathbf{false} \rhd e_2 \ =_{df} \ e_2$$

Being able to control the flow of the program, we are now interested in describing the effects of a list of predicates once executed in sequence. Let P and Q be predicates describing the behaviour of two programs, the construct $P; Q$ describes the program that first executes P and, when P has finished, starts Q. The key aspect of this way of composing programs is that there is an intermediate state, just after P finishes and before Q starts, where the final state of P is passed on as the initial state of Q. This intermediate state should not be observable, as it is just a "stepping stone" that links the execution of P with the one of Q. The UTP definition of this operator captures this notion by existentially quantifying this intermediate step:

Definition 2.2.3. *Sequential composition*

$$P; Q \ =_{df} \ \exists v_0 \bullet P[v_0/v'] \wedge Q[v_0/v]$$

provided v_0 *is a fresh variable and* $out\alpha(P) = in\alpha(Q) = v$

Regardless of the complexity of its definition, sequential composition satisfies the expected algebraic laws of associativity and distributivity through the conditional construct.

Law 2.2.4. $P;(Q;R) = (P;Q);R$

Law 2.2.5. $(P \lhd b \rhd Q);R = (P;R) \lhd b \rhd (Q;R)$

We have presented ways of combining programs by means of the conditional and sequential composition constructs, but we have not explained the elements that programs are constructed from. In the imperative paradigm, assignment is the most basic action a program can perform. An assignment of the form $x := e$ causes the final value of x to be equal to the value of expression e. As we use dashed variables to refer to the after value of a variable, the effect of the above assignment can be described as $x' = e$.

The informal definition above is correct in the sense it matches one side of the operational intuition of an assignment: to update the value of the appropriate variable. On the other hand, we also expect an assignment to leave all other variables unchanged. We have already mentioned that one of the distinguishing features in the UTP is that every predicate has an alphabet. In the case of the assignment construct, the alphabet allows us to state the fact that all variables not mentioned in the assignment remain unchanged. More formally, assignment is defined as follows:

Definition 2.2.6. *Assignment*

$$x := e \ =_{df} \ (x' = e \land v' = v)$$

where $\alpha(x := e) = \{x, v\}$

It is useful, mostly for reasoning about languages and programs, to have a command that has no effect. This command always terminates and leaves the value of all variables unchanged. For a given alphabet $A = \{v, v'\}$, we can easily express the semantics of the construct with no effect as the assignment $v := v$. If we denote this construct with the symbol $I\!I$, we can define its meaning as follows.

Definition 2.2.7. *Skip*

$$I\!I_A \ =_{df} \ (v' = v)$$

The main characteristic of $I\!I$ (i.e., having no effect whatsoever over the program) is precisely captured by the fact it is the unit for sequential composition:

Law 2.2.8. *Provided P terminates we have:*

$$P;I\!I_{\alpha P} = P = I\!I_{\alpha P};P$$

On the other hand, the non-deterministic choice $P \sqcap Q$ stands for a program that may behave either like P or like Q, the choice being totally arbitrary. In the UTP, this behaviour between two predicates can be simply described as disjunction.

Definition 2.2.9. *Non-deterministic choice*

$$P \sqcap Q \ =_{df} \ P \lor Q$$

provided $\alpha(P) = \alpha(Q)$

Another distinguishing feature of the UTP is its concern with correctness in the program development process. In this regards, "every possible observation of any run of the program will yield values which make the specification true" [Hoare and He 1998, Chapter 1.5]. In other words, the behaviour of an implementation always implies its specification. If we denote universal quantification over all variables in the alphabet by means of surrounding the quantified expression by square brackets we find that:

Definition 2.2.10. *Refinement*

$$P \sqsubseteq Q \quad \textit{if and only if} \quad [Q \Rightarrow P]$$

From the above definition it is easy to see that the refinement ordering is a partial order: reflexive, anti-symmetric, and transitive. Moreover, the set of alphabetised predicates with a particular alphabet A is a complete lattice [Davey and Priestley 2002] under the refinement ordering. The definition of refinement also allows us to prove a key result in the refinement calculus: that reducing non-determinism leads to refinement.

Law 2.2.11. $P \sqcap Q \sqsubseteq P$

It follows from the result above that a component P can be made less deterministic by adding the possibility of it behaving like Q. An increase in a program's non-determinism is, in general, undesirable as it makes the program less predictable, more difficult to control and, ultimately, more likely to go wrong. Pushing this argument to the extreme, the worst component of all is the one that is totally unpredictable. This program can only be captured by the weakest predicate **true** (with alphabet A): this is the program that aborts and behaves arbitrarily.

Definition 2.2.12. *Abort*

$$\bot =_{df} \textit{true}$$

It is straightforward to show that abort is the weakest element in the implication ordering; hence it is the bottom element of the predicate lattice ordered by implication (refinement). The top element, on the other hand, is denoted $\top$, and is the strongest predicate **false**: this is the program that performs miracles and implements every specification.

Definition 2.2.13. *Miracle*

$$\top =_{df} \textit{false}$$

The fact that abort and miracle are, respectively, the bottom and top elements of the predicate lattice are captured by the following two laws, which hold for all P with alphabet A.

Law 2.2.14. $\bot \sqsubseteq P$

Law 2.2.15. $P \sqsubseteq \top$

In turn, the least upper bound operator in our predicate lattice is not defined in terms of the relational model, but by law 2.2.16 below.

Law 2.2.16. $P \sqsubseteq (\bigsqcap S)$ ***if and only if*** $(P \sqsubseteq X$ ***for all*** $X \in S)$

the dual of the operator defined above is defined over the conjunction of all relations in a set S denoted $\bigsqcup S$ and describes the least upper bound of a set S. Its definition is by means of the dual of Law 2.2.16 with the implications in the opposite direction:

Law 2.2.17. $(\bigsqcup S) \sqsubseteq P$ ***if and only if*** $(X \sqsubseteq P$ ***for all*** $X \in S)$

The final programming feature we are interested in discussing is scopes for program variables. Variable blocks are split into the commands **var** x, which declares and introduces x in scope, and **end** x, which, in turn, removes x from scope. Their definitions are presented below, where A is an alphabet containing x and x'.

Definition 2.2.18.

$$\textbf{\textit{var}}\ x =_{df} \exists x \bullet \mathbb{I}_A$$

Definition 2.2.19.

$$\textbf{\textit{end}}\ x =_{df} \exists x' \bullet \mathbb{I}_A$$

From this definition and the notion of sequential composition, it is easy to show that variable declaration and undeclaration act like existential quantification over their scopes, as shown by the results below.

Law 2.2.20. $\textbf{\textit{var}}\ x; P =_{df} \exists x \bullet P$

Law 2.2.21. $P; \textbf{\textit{end}}\ x =_{df} \exists x' \bullet P$

Up to this point we have presented a theory capable of describing most of the constructs present in languages belonging to the imperative-sequential paradigm. Our goal is now to define the meaning to be assigned to diverging behaviour. The natural choice at this point would be to associate a diverging program with abort (the worst possible predicate in the theory). This definition achieves its purpose in the sense that the meaning of a diverting loop is completely non-deterministic. Unfortunately, it is not possible to prove that $\bot$ is a left zero for sequential composition (i.e., Law $\bot; P = \bot$) the theory of alphabetised propositional calculus. In the context of the semantics for iteration, this means the theory allows programs to recover from, for example, a non-terminating loop. This result clearly contradicts reality for sequential programming, where a non-terminating loop will iterate for ever and never recover to be able to produce valid computations again.

Unfortunately, there is no clear way to address this problem in the general framework of alphabetised predicates. The solution is to restrict ourselves to a subset theory, where we are only interested in predicates of the form $(P \vdash Q)$ where P and Q are the program's precondition and postcondition respectively. The next section addresses this theory in detail.

2.3 The design theory

As mentioned in the previous section, the theory of alphabetised relations has the problem that it allows, for example, non-terminating loops to be ignored. The solution to this problem in the UTP is to restrict the theory to a subset of the alphabetised predicates in which we are able to reason about termination. In the new theory, a new observational variable, called ok, is used to record information about the start and termination of programs. ok records that the program has started, and ok' records that it has terminated. These are *special* variables, in the sense that they are included in the design's alphabet, but they cannot appear in code or in preconditions and postconditions.

Predicates satisfying the restrictions above are called *designs* and the main advantage of having predicates in the design form is that they can be split into precondition–postcondition pairs. When implementing a design, we are allowed to assume that the precondition holds and the program has started, but we have to fulfil the postcondition and ensure the program terminates. In the case the precondition does not hold or the program was not started, we are neither committed to establish the postcondition nor to make the program terminate. A design with precondition P and postcondition Q, for predicates P and Q not containing ok or ok', is written $(P \vdash Q)$ and it is defined as follows.

Definition 2.3.1. *Design*

$$(P \vdash Q) =_{df} (ok \wedge P \Rightarrow ok' \wedge Q)$$

2.3.1 Miracle, abort and refinement revisited

In the design theory, abort has precondition **false** and is never guaranteed to terminate:

Definition 2.3.2. *The design abort*

$$\bot_D =_{df} (\textbf{\textit{false}} \vdash \textbf{\textit{true}})$$

Notice that because of the underlying implication in the definition of designs, there are infinite characterisations for abort in the design theory (as **false** $\Rightarrow P$ for any predicate P). However, only two of them are taken as the definition of the design abort, as shown in the following definition.

Theorem 2.3.3. *Characterisation of abort*

$$\bot_D = (\textbf{\textit{false}} \vdash P)$$

for any predicate P

An immediate consequence of the above definition is that abort becomes a left zero for sequential composition. In this way the problem of the alphabetised relational theory does not exist in the design theory. This fact is summarised by the following law.

Law 2.3.4. $\bot_D ; P = \bot_D$

On the other hand, the miraculous program in the design theory has precondition **true**, and establishes the impossible: **false**. In fact, as it achieves the impossible, the miraculous design is the design that cannot be started.

Theorem 2.3.5. *The miraculous design*

$$\top_D =_{df} (\textit{true} \vdash \textit{false}) = \neg ok$$

As mentioned before, the concept of refinement, defined as the implementation implying the specification, remains constant in all UTP theories described in this thesis. An interesting (and reassuring) result about the design theory is the fact that refinement amounts to either weakening the precondition, or strengthening the postcondition in the presence of the precondition. This is established by the result below.

Theorem 2.3.6. *Design refinement*

$$(Q_1 \vdash Q_2) \sqsubseteq (P_1 \vdash P_2) \quad \textit{if and only if} \quad [Q_1 \Rightarrow P_1] \wedge [(Q_1 \wedge P_2) \Rightarrow Q_2]$$

2.3.2 Healthiness conditions

We can identify up to four healthiness conditions on a program P in the design theory: **H1** requires that observations cannot be made before P has started; **H2** states that P cannot require non-termination, (i.e., it is always possible for P to terminate); **H3** requires P's precondition to be just a condition (instead of a predicate); and **H4** imposes the restriction that for every initial value of the observational variables on the input alphabet, there exist final values for the variables of the output alphabet (i.e., P is feasible). Their precise characterisation is presented in the definition below:

Definition 2.3.7.

H1 $P = (ok \Rightarrow P)$

H2 $[P[\textit{false}/ok'] \Rightarrow P[\textit{true}/ok']]$

H3 $P = P; \mathbb{I}_D$

H4 $P; \bot_D = \bot_D$

The interpretations presented above are clear from the definitions of the healthiness conditions and the expansions of the operators involved. For further information regarding the details of this claim, refer to [Hoare and He 1998, Chapter 3].

2.3.3 Assignment, skip, sequential composition and conditional

In this new setting, it is necessary to redefine assignment and skip, as the definitions introduced in the previous section are not designs. In both cases, the definition as a design has precondition **true** and the propositional version of the construct as postcondition. More formally, we define the design skip and assignment as follows.

Definition 2.3.8. *Design skip*

$$\mathit{II}_D =_{df} (\textbf{\textit{true}} \vdash \mathit{II})$$

Definition 2.3.9. *Design assignment*

$$x := e =_{df} (\textbf{\textit{true}} \vdash x' = e \wedge v' = v)$$

As in the alphabetised relational calculus, II_D is the left unit for sequential composition. Unfortunately, it is not always the case that II_D is also the right unit for sequential composition. Only **H3** designs satisfy that II_D is their unit for sequential composition[1].

Law 2.3.10. $\mathit{II}_D ; P = P$

The sequential composition of two assignments to the same variable is easily combined into a single assignment.

Law 2.3.11. $x := e ; x := f(x) = x := f(e)$

In the case of assignment, sequential composition has $\bot_D$ and $\top_D$ as right zeroes and II_D as right unit.

Law 2.3.12. $x := e ; \bot_D = \bot_D$

Law 2.3.13. $x := e ; \top_D = \top_D$

Law 2.3.14. $x := e ; \mathit{II}_D = x := e$

It is also possible to commute the order of a sequence of assignments provided they do not depend on each other.

Law 2.3.15. *If e_1 does not depend on y and e_2 does not depend on x then:*

$$(x := e_1 ; y := e_2) = (y := e_2 ; x := e_1)$$

Assignment to multiple variables is defined as the simultaneous update of each of the individual variables involved. Multiple assignment is commutative and, when updating different variables, it can also be expressed as a sequence of individual assignments

Law 2.3.16. $(x, y := e_1, e_2) = (y, x := e_2, e_1)$

Law 2.3.17. *If $x \neq y$ and e_2 does not mention x we have that:*

$$(x := e_1 ; y := e_2) = (x, y := e_1, e_2)$$

[1] To keep the presentation compact, we have decided to only postulate the laws and theorems in this part of the thesis. The proof of the laws and theorems from this and all subsequent chapters can be found in the appendices at the end of the thesis.

All program operators from the previous section are closed under the design theory (i.e., when applied to designs the result is also a design). The theorems below capture this fact for the conditional and sequential composition combinators. In the case of the conditional construct, when the choice between two designs depends on a condition b, then so do the precondition and the postcondition of the resulting design.

Theorem 2.3.18. *Design conditional*

$$(P_1 \vdash P_2) \lhd b \rhd (Q_1 \vdash Q_2) = (P_1 \lhd b \rhd Q_1 \vdash P_2 \lhd b \rhd Q_2)$$

A sequence of designs $(P_1 \vdash P_2)$ and $(Q_1 \vdash Q_2)$ only terminates when P_1 is feasible and P_2 is guaranteed to establish Q_1. On termination, it establishes the sequential composition of the postconditions.

Theorem 2.3.19. *Design sequential composition*

$$(P_1 \vdash P_2); (Q_1 \vdash Q_2) = (\neg(\neg P_1; \mathit{true}) \wedge \neg(P_2; \neg Q_1) \vdash P_2; Q_2)$$

The selection between two designs based on a condition satisfies a number of familiar algebraic laws. The most basic property of a conditional is that its left branch is executed if the condition is known to hold; otherwise, the other branch is selected.

Law 2.3.20. $P \lhd \mathit{true} \rhd Q = P$

Law 2.3.21. $P \lhd \mathit{false} \rhd Q = Q$

If both branches of the selection construct are the same program the choice is irrelevant and the conditional can be eliminated.

Law 2.3.22. $P \lhd b \rhd P = P$

The conditional construct satisfies a special form of the commutativity property: we can swap the branches but the condition upon which we are performing the selection needs to be inverted.

Law 2.3.23. $P \lhd b \rhd Q = Q \lhd \neg b \rhd P$

The sequential composition operator distributes leftward through the conditional.

Law 2.3.24. $(P \lhd b \rhd Q); S = (P; S) \lhd b \rhd (Q; S)$

In the case where an assignment is followed by a conditional, we can push the assignment inside both branches of the conditional provided we make sure the condition is evaluated with the value the assignment was modifying.

Law 2.3.25. $x := e; P \lhd b(x) \rhd Q = (x := e; P) \lhd b(e) \rhd (x := e; Q)$

The selection between two assignments to the same variable can be reduced to a single assignment where the condition has been pushed into the expression on the right hand side.

Law 2.3.26. $(x := e_1 \lhd b \rhd x := e_2) = x := (e_1 \lhd b \rhd e_2)$

The following laws allow the simplification of nested conditionals.

Law 2.3.27. $(P \lhd b \rhd Q) \lhd b \rhd S = P \lhd b \rhd S = P \lhd b \rhd (Q \lhd b \rhd S)$

Law 2.3.28. $(P \lhd b \rhd Q) \lhd c \rhd R = P \lhd b \wedge c \rhd (Q \lhd c \rhd R)$

Law 2.3.29. $P \lhd b \rhd (Q \lhd c \rhd R) = (P \lhd b \rhd Q) \lhd c \rhd (P \lhd b \rhd R)$

Law 2.3.30. $(P \lhd b \rhd Q) \lhd c \rhd R = (P \lhd c \rhd R) \lhd b \rhd (Q \lhd c \rhd R)$

Law 2.3.31. $P \lhd b \rhd (P \lhd c \rhd Q) = P \lhd (b \vee c) \rhd Q$

Law 2.3.32. $P \lhd (s \wedge b) \rhd (Q \lhd (s \wedge \neg b) \rhd R) = P \lhd (s \wedge b) \rhd (Q \lhd s \rhd R)$

Regarding the sequential composition of designs, it is easy to show it is associative and it also has miracle as a left zero.

Law 2.3.33. $P; (Q; R) = (P; Q); R$

Law 2.3.34. $\top_D; P = \top_D$

2.3.4 Scope and program variables

Even though the scope delimiters **var** and **end** are the same in the design theory as they were in the alphabetised predicate calculus (i.e., existential quantification of the undashed and dashed variable of interest respectively), it is possible express them as designs. The following definitions can be shown to be equivalent to Definitions 2.2.18 and 2.2.19 respectively.

Definition 2.3.35. *Design start of scope*

$$\textbf{\textit{var}}\ x\ =_{df}\ (\textbf{\textit{true}} \vdash \textbf{\textit{var}}\ x)$$

Definition 2.3.36. *Design end of scope*

$$\textbf{\textit{end}}\ x\ =_{df}\ (\textbf{\textit{true}} \vdash \textbf{\textit{end}}\ x)$$

As expected, opening and immediately closing the scope of a variable has no effect whatsoever over the program. On the other hand, closing the scope of an existing variable and immediately opening it again produces a loss of information (and hence, increases the non-determinism).

Law 2.3.37. $\textbf{\textit{var}}\ x; \textbf{\textit{end}}\ x = \mathit{II}$

Law 2.3.38. $\textbf{\textit{end}}\ x; \textbf{\textit{var}}\ x \sqsubseteq \mathit{II}$

The results from Law 2.3.38 can be further strengthened provided we follow the closing and opening of the scope of a variable x with an independent assignment to it.

Law 2.3.39. *If x is not free in e then we have that:*

$$(\textbf{\textit{end}}\ x; \textbf{\textit{var}}\ x := e) = (x := e)$$

An assignment to a variable just before the end of its scope is completely irrelevant.

Law 2.3.40. $(x := e; \textbf{\textit{end}}\ x) = \textbf{\textit{end}}\ x$

The effect of introducing a variable in scope is to incorporate the variable name to the alphabet and also to initialise the variable to an arbitrary value. This non-determinism can be reduced by initialising the variable after its declaration.

Law 2.3.41. $\textbf{\textit{var}}\ x \sqsubseteq \textbf{\textit{var}}\ x; x := e$

A sequence of variable declarations (undeclarations) can be subsumed into a single declaration (undeclaration) of the list of variables.

Law 2.3.42. $\textbf{\textit{var}}\ x; \textbf{\textit{var}}\ y = \textbf{\textit{var}}\ x, y$

Law 2.3.43. $\textbf{\textit{end}}\ x; \textbf{\textit{end}}\ y = \textbf{\textit{end}}\ x, y$

The order of declaration (undeclaration) of variables is irrelevant.

Law 2.3.44. $\textbf{\textit{var}}\ x, y = \textbf{\textit{var}}\ y, x$

Law 2.3.45. $\textbf{\textit{end}}\ x, y = \textbf{\textit{end}}\ y, x$

Up to this point we have presented a mechanism that when given a program P that contains x and x' in its alphabet, allows us to remove them from the alphabet. Some times it is necessary to be able to perform the converse operation: to be able to produce a program that does have x or x' in its alphabet from a program Q that does not contain them. Since Q does not update the value of x, it makes sense to assume the value of x is kept constant. The following definition captures this notion.

Definition 2.3.46. *Alphabet extension*

$$(P_1 \vdash P_2)_{+x} =_{df} (P_1 \vdash P_2 \wedge x' = x)$$

provided $x \notin \alpha(P_1 \vdash P_2)$

With the definition above we can now introduce the following laws that describe the process of expanding a variable's scope over a process that does not mention the variable.

Law 2.3.47. *If P does not mention x then*

$$P; \textbf{\textit{var}}\ x = \textbf{\textit{var}}\ x; P_{+x}$$

Law 2.3.48. *If P does not mention x then:*

$$\textbf{\textit{end}}\ x; P = P_{+x}; \textbf{\textit{end}}\ x$$

It is possible to end the scope of a variable x just before an assignment to it, provided the value assigned does not depend on the value of x.

Law 2.3.49. *Provided P is **H3** and e does not mention x we have:*

$$P; (x := e) = (P; \textbf{\textit{end}}\ x)_{+x}; (x := e)$$

2.3.5 Assertions and assumptions

The assumption of a condition b, denoted by $b^\top$ can be regarded as conditional where: the whole construct terminates immediately leaving the state unchanged (i.e., it behaves like skip) if the condition is **true**, or it behaves like miracle otherwise.

Definition 2.3.50. *Assumption*

$$b^\top =_{df} \mathit{II}_D \lhd b \rhd \top_D$$

On the other hand, the assertion $b_\perp$ also behaves like skip when b holds, but it behaves like abort otherwise.

Definition 2.3.51. *Assertion*

$$b_\perp =_{df} \mathit{II}_D \lhd b \rhd \perp_D$$

By introducing these two constructs, the UTP theory allows assertional reasoning within any language that can be given semantics using the theory of designs. This is one of the great advantages of the UTP as a unifying mechanism for programming languages.

Up to this point we have defined the assumption and assertion construct in terms of more basic constructs. In certain circumstances it will be useful to be able to express assertions and assumptions as a design. The following two theorems provide the design form for these two constructs.

Theorem 2.3.52. *Design characterisation of assumption*

$$b^\top = (\textbf{\textit{true}} \vdash \mathit{II} \lhd b \rhd \textbf{\textit{false}})$$

Theorem 2.3.53. *Design characterisation of assertion*

$$b_\perp = (b \vdash \mathit{II})$$

Assuming a **false** condition leads to miraculous behaviour, while asserting a false condition leads to abortion.

Law 2.3.54. $\textbf{\textit{false}}^\top = \top_D$

Law 2.3.55. $\textbf{\textit{false}}_\perp = \perp_D$

The following laws allow the simplification of sequences of assertions and assumptions within our language.

Law 2.3.56. $b^\top ; b^\top = b^\top$

Law 2.3.57. $b_\bot ; b_\bot = b_\bot$

Law 2.3.58. $b_\bot ; b^\top = b_\bot$

Law 2.3.59. $b^\top ; b_\bot = b^\top$

Law 2.3.60. $b^\top ; c^\top = (b \wedge c)^\top$

Law 2.3.61. $b_\bot ; c_\bot = (b \wedge c)_\bot$

Law 2.3.62. $(b \vee c)^\top ; b^\top = b^\top$

Also, the sequential composition of assumptions commutes.

Law 2.3.63. $b^\top ; c^\top = c^\top ; b^\top$

Assertion is refined by skip and skip is, in turn, refined by assumption[2].

Law 2.3.64. $b^\top \sqsupseteq I\!I_D \sqsupseteq b_\bot$

Asserting or assuming the value of a variable x is equal to an expression e has no effect if they are performed just after an assignment setting x to that value.

Law 2.3.65. *If e_1 does not mention x then:*

$$(x, y := e_1, e_2) = (x, y := e_1, e_2); (x = e_1)_\bot$$

Law 2.3.66. *If e_1 does not mention x then:*

$$(x, y := e_1, e_2) = (x, y := e_1, e_2); (x = e_1)^\top$$

Assumption and assertion commute with assignment provided their condition does not depend on the value modified by the assignment.

Law 2.3.67. *If b does not depend on x then*

$$x := e; b^\top = b^\top ; x := e$$

Law 2.3.68. *If b does not depend on x then*

$$x := e; b_\bot = b_\bot ; x := e$$

[2]From this observation it is possible to conclude that $(b_\bot, b^\top)$ is a simulation in line with [Hoare and He 1998, Chapter 4]

It is also possible to shift an assertion about the value of a variable x before and after a more general predicate P, provided it keeps the value of x constant.

Law 2.3.69. *Provided P is **H3** and **H4** we have:*

$$P_{+x}; (x = e)_\perp = (x = e)_\perp; P_{+x}$$

Two variables that have been made equal by an assignment can be exchanged when used in the condition of an assertion.

Law 2.3.70. $x := y; (y = e)_\perp = x := y; (x = e)_\perp$

It is possible to assume the value of the condition inside the branches of a conditional.

Law 2.3.71. $P \lhd b \rhd Q = (b^\top; P) \lhd b \rhd ((\neg b)^\top; Q)$

The following laws allow the simplification of the conditional construct based on knowledge provided by assertions or assumptions.

Law 2.3.72. $b^\top; (P \lhd b \rhd Q) = b^\top; P$

Law 2.3.73. $(\neg b)_\perp; (P \lhd b \vee c \rhd Q) = (\neg b)_\perp; (P \lhd c \rhd Q)$

Law 2.3.74. $b^\top; (P \lhd b \vee c \rhd Q) = b^\top; P$

Law 2.3.75. $(\neg b)^\top; (P \lhd b \rhd Q) = (\neg b)^\top; Q$

Law 2.3.76. $b_\perp; (P \lhd c \rhd Q) = (b_\perp; P) \lhd c \rhd (b_\perp; Q)$

Law 2.3.77. $b^\top; (P \lhd c \rhd Q) = (b^\top; P) \lhd c \rhd (b^\top; Q)$

Law 2.3.78. *Provided* $\neg(b \wedge c)$ *we have:*

$$b^\top; (P \lhd c \rhd Q) = b^\top; Q$$

To introduce a variable followed by an assumption about the value it holds is equivalent to introducing the variable followed by an assignment of that value to the variable.

Law 2.3.79. ***var*** $x; (x = e)^\top =$ ***var*** $x; x := e$

2.3.6 Recursion and iteration

If X stands for the recursive program we are constructing and $F(X)$ describes the behaviour of the program for a given context F, then $\mu X \bullet F(X)$ is a solution to the equation $X = F(X)$. Furthermore, it is the least solution. The following two laws capture these notions.

Law 2.3.80. $\mu X \bullet F(X) = F(\mu X \bullet F(X))$

Law 2.3.81. $F(Y) \sqsubseteq Y \Rightarrow \mu X \bullet F(X) \sqsubseteq Y$

The construct $(b * P)$ is a syntactic abbreviation of the more conventional iteration construct

```
while b do P
```

and it is defined as the least fixed point of the equation: $(P; X) \lhd b \rhd \mathit{II}_D$. More formally:

Definition 2.3.82. *Loop*

$$b * P = \mu X \bullet (P; X) \lhd b \rhd \mathit{II}_D$$

If the loop's condition does not hold at the beginning of its execution, it reduces to skip. On the other hand, if the condition does hold, its body gets executed at least once.

Law 2.3.83. $(\neg b)^\top; b * P = (\neg b)^\top$

Law 2.3.84. $(\neg b)_\bot; b * P = (\neg b)_\bot$

Law 2.3.85. $b^\top; b * P = b^\top; P; (b * P)$

When a loop terminates, its condition is necessarily false.

Law 2.3.86. $b * P = (b * P); (\neg b)_\bot$

If P preserves a certain condition, then so does $b * P$.

Law 2.3.87. *Provided* $c_\bot; P = c_\bot; P; c_\bot$ *we have:*

$$c_\bot; (b * P) = c_\bot; (b * P); c_\bot$$

2.3.7 Disjoint-alphabet parallelism

Parallel composition is initially defined in the UTP as the conjunction of the predicates describing the behaviour of the individual components that are placed in parallel [Hoare and He 1998, Chapter 1]. In order to avoid interference between the parallel processes it is necessary for the variables in each of the parallel processes' alphabet to be different from each other. This notion is known as *alphabet disjointness* and it is defined as follows:

Definition 2.3.88. *Disjoint alphabets*

$$\alpha P \cap \alpha Q = \emptyset$$

A consequence of modelling concurrency as conjunction is that it allows each process to start and finish independently from each other. This is not a very desirable feature as parallel programming usually requires synchronisation on start and termination. A consequence of the synchronicity at the beginning and end of the parallel execution is that if one of the parallel processes diverges, then the whole parallel program diverges. Unfortunately, capturing parallel composition as conjunction fails at capturing this behaviour.

The solution is in the design theory and its finer degree of control over termination. The parallel composition of two disjoint designs P and Q is defined as:

Definition 2.3.89. *Disjoint alphabet parallelism*

$$(P_1 \vdash P_2) \parallel (Q_1 \vdash Q_2) \;=_{df}\; (P_1 \wedge Q_1 \vdash P_2 \wedge Q_2)$$

provided $\alpha P \cap \alpha Q = \emptyset$, $P = (P_1 \vdash P_2)$ *and* $Q = (Q_1 \vdash Q_2)$.

Disjoint-alphabet parallelism is commutative, associative and it has unit $I\!I_D$ and zero $\bot_D$.

Law 2.3.90. $P \parallel Q = Q \parallel P$

Law 2.3.91. $P \parallel (Q \parallel R) = (P \parallel Q) \parallel R$

Law 2.3.92. $(I\!I_A \parallel P) = P_{+A}$

Law 2.3.93. $\bot_D \parallel P = \bot_D$

Disjoint-alphabet parallelism distributes over conditional.

Law 2.3.94. $(P \lhd b \rhd Q) \parallel R = ((P \parallel R) \lhd b \rhd (Q \parallel R))$

For the following law we need to introduce the notion of a descending chain: an infinite disjunction of weakening predicates, generally used to model the process of successive approximation. More formally, a set of predicates $E = \{E_i \mid i \in \mathbb{N}\}$ is a descending chain for a predicate P provided

$$E_0 = \textbf{false}$$
$$[E_i \sqsupseteq E_{i+1}] \textbf{ for all } i \in \mathbb{N}$$
$$P = \bigvee_i E_i$$

Taking advantage of the definition above, it is also possible to show that disjoint-alphabet parallelism distributes over least upper bounds.

Law 2.3.95. *For any descending chain* $S = \{S_n \mid n \in \mathbb{N}\}$ *we have that:*

$$(\bigsqcup S) \parallel R = \bigsqcup_n (S_n \parallel R)$$

Sequential composition can be exchanged with disjoint-alphabet parallelism under certain conditions. This law essentially states the order of application of parallel and sequential composition is irrelevant to the final result. This property is sometimes called the *abides property*[3].

Law 2.3.96. *If both P and Q are* ***H4*** *we have that:*

$$(P; R) \parallel (Q; S) = (P \parallel Q); (R \parallel S)$$

[3]The word "abides" can be seen as a contraction of "above-besides", which gives the 2-dimensional interpretation of this property where parallel and sequential composition are performed horizontally and vertically respectively. In this setting, the property states that any order or execution (i.e., rows first followed by columns or *vice versa*) will produce the same result.

The disjoint-alphabet parallel operator can be reduced to sequential composition, provided the right alphabet extensions are in place.

Law 2.3.97. *If P and Q are **H3** and **H4**, then:*

$$P \parallel Q = P_{+\alpha Q}; Q_{+\alpha P} = Q_{+\alpha P}; P_{+\alpha Q}$$

The parallel execution of two individual assignments is equivalent to a multiple assignment.

Law 2.3.98. $(x := e_1 \parallel y := e_2) = x, y := e_1, e_2$

The scope of a variable can be reduced to just one of the parallel branches (provided the other branch does not mention that variable).

Law 2.3.99. *Provided Q does not mention x we have:*

$$\textbf{\textit{var}}\ x; (P \parallel Q) = (\textbf{\textit{var}}\ x; P) \parallel Q$$

Law 2.3.100. *Provided Q does not mention x and P and Q are **H3**, we have:*

$$(P \parallel Q); \textbf{\textit{end}}\ x = (P; \textbf{\textit{end}}\ x) \parallel Q$$

2.3.8 Parallel by merge

So far we have presented a form of parallelism that relies on the parallel processes being disjoint. The assumption of disjointness enables a very simple definition for the parallel operator that can be implemented by means of real parallel execution or by arbitrary interleaving of the parallel processes. Nevertheless, it is rarely the case that we are in a disjoint-variable environment when we are dealing with commercial programming languages or hardware.

With shared-variables, the parallel composition operator requires a mechanism for dealing with the effects of simultaneous updates to the same variables. In the UTP, this is achieved by means of creating local copies of the shared variables for each of the parallel processes. In this way, each process can freely modify their own private copy of the shared store without the risk of inconsistencies. When both processes have terminated, the local copies are joined by a merge predicate that calculates the effect on the shared variables from the values in their local copies.

The creation of local copies of a list of shared variables m is achieved by substituting m' by $i.m'$ (where i is an index), transforming a parallel process of the form $P(m, m')$ into the renamed $P(m, i.m')$. In this way, the output alphabet of P no longer mentions m', hence it cannot interfere with other parallel processes modifying m'. The substitution described above is achieved by means of a *separating simulation*, the definition of which is as follows.

Definition 2.3.101. *Separating simulation*

$$Ui(m) =_{df} \textbf{\textit{var}}\ i.m := m; \textbf{\textit{end}}\ m$$

From the above definition it is possible to identify a number of useful algebraic properties about the separating simulation predicate. Firstly, the order in which the shared variables are renamed is not relevant. Moreover, the separation of a list of variables can be done in a single step or one at a time. The following two laws characterise these two properties.

Law 2.3.102. *Provided x and y are different variables we have:*

$$U0(x, y) = U0(y, x)$$

Law 2.3.103. *Provided x and y are different variables we have:*

$$U0(x, y) = U0(x); U0(y)$$

The separating simulation construct defined before allows us to rename the variables of parallel processes so they can be combined with the disjoint-alphabet parallel operator. The missing step is the mechanism for merging the final value of the local copies that each process was modifying. The merging predicate is usually denoted by M and it updates m from the values in $0.m$ and $1.m$. In this context, the definition of *parallel by merge* is as follows:

Definition 2.3.104. *Parallel by merge*

$$P \parallel_M Q =_{df} ((P; U0(m)) \parallel (Q; U1(m))); M$$

The M predicate is clearly domain specific, as each domain will require a different way of calculating the final value for the shared variables from the local copies. Instead of treating each possible definition, Hoare and He [1998, Chapter 7] present a set of *validity properties* for M.

Definition 2.3.105. *Valid merge*

1. *M is symmetric in its input* $0.m$ *and* $1.m$*:*

$$(0.m, 1.m := 1.m, 0.m); M = M$$

2. *M is associative*

$$(0.m, 1.m, 2.m := 1.m, 2.m, 0.m); M3 = M3$$

provided $M3 = \exists x, t \bullet M(ok, m, 0.m, 1.m, x, t) \wedge M(t, m, x, 2.m, m', ok')$

3. *M is idempotent in its input* $0.m$ *and* $1.m$

$$(0.m, 1.m := m, m); M = I\!I$$

The validity requirements capture the minimal set of properties a merge function M must satisfy if commutativity, associativity and $I\!I$-identity are to be proved for a parallel by merge operator based on M. For further details, the reader is referred to [Hoare and He 1998, theorem 7.2.10].

As mentioned above, provided M is a *valid merge*, a set of laws similar to the one satisfied by disjoint-alphabet parallel operator can be proved for the parallel by merge operator. Disjoint-alphabet parallelism is commutative, associative and it has pseudo-unit Ⅱ_D and zero $\perp_D$.

Law 2.3.106. $P \parallel_M Q = Q \parallel_M P$

Law 2.3.107. $P \parallel (Q \parallel_M R) = (P \parallel_M Q) \parallel_M R$

Law 2.3.108. $(\text{Ⅱ}_A \parallel_M P) \sqsubseteq P_{+A}$

Law 2.3.109. $\mathit{true} \parallel_M P = \mathit{true}$

Disjoint-alphabet parallelism distributes over conditional and least upper bounds.

Law 2.3.110. $(P \lhd b \rhd Q) \parallel_M R = ((P \parallel R) \lhd b \rhd (Q \parallel R))$

Law 2.3.111. *For any descending chain* $S = \{S_n \mid n \in \mathcal{N}\}$ *we have that:*

$$(\bigsqcap S) \parallel_M R = \bigsqcap_n (S_n \parallel_M R)$$

A restricted version of the abudes principle holds for parallel by merge.

Law 2.3.112. *Provided* $x := e$ *does not mention* m *we have that:*

$$(x := e; P) \parallel_M Q = (x := e); (P \parallel_M Q)$$

Regarding the healthiness conditions in the design theory, the following theorem shows that provided M preserves healthiness conditions **H1** to **H4**, then the healthiness conditions that P and Q satisfy will be preserved by the parallel by merge operator.

Theorem 2.3.113. $\parallel_M$ *preserves* ***H1*** *to* ***H4****.*

2.3.8.1 The merge predicate

The goal of this section is to define a particular merge operator for Handel-C programs. We will also show our operator is *valid* (i.e., it satisfies the *valid merge* criteria from the UTP).

We take advantage of two domain-specific facts about Handel-C: (a) its synchronous nature ensures that M will be joining the results of two parallel processes that perform their actions within a single clock cycle; and (b) Handel-C semantics allows at most one write to any shared variable per clock cycle[4]. Our definition of M merges the results from $P \parallel_M Q$ by updating the shared variable m with the value in the local copies of m that changed during the parallel execution of P and Q. If none of the parallel processes modified its local copy of m, then M keeps m constant.

[4]This is a natural restriction given the hardware-oriented nature of Handel-C and the fact that flip-flops do not allow simultaneous writes.

Definition 2.3.114. *Merge operator*

$$M(ok, m, 0.m, 1.m, m', ok') =_{df} (ok' = ok) \wedge m' = \left(\begin{array}{l} (m \lhd m = 1.m \rhd 1.m) \\ \quad \lhd m = 0.m \rhd \\ (0.m \lhd m = 1.m \rhd (1.m \sqcap 0.m)) \end{array} \right)$$

From the above definition it is easy to show that M is symmetric and that it reduces to skip when both of the copies to be merged are equal to the original value for the shared variable. For the rest of this thesis we will sometimes omit some/all of the arguments passed to M when they are clear from the context. Furthermore, we will not mention ok and ok' any further in this thesis, as they correspond to the theory's observational variables with the same name.

Law 2.3.115. $(0.m, 1.m := 1.m, 0.m); M = M$

Law 2.3.116. $(0.m, 1.m := m, m); M = I\!I$

From Definition 2.3.114, it is important to notice that we have "totalised" the definition of M in order to cover the (impossible) case where the two parallel processes modify m. According to our definition, the result of multiple assignments to the same variable during the same clock-cycle is the internal choice of updating the store with either of the values being assigned. This unexpected non-determinism can be explained at the hardware level by the unpredictable value that will be stored in a register when it is fed with more than one value at the same time. The consequence of this "extended" behaviour allows M to be associative on its input, as shown by the following result.

Law 2.3.117. $(0.m, 1.m, 2.m := 1.m, 2.m, 0.m); M3 = M3$

From the three results above, M is a *valid merge* predicate. A shared-variables parallel operator with our merge predicate has the properties defined in the previous section. In addition to the validity properties, M also satisfies the following two laws:

Law 2.3.118. $(0.m = v); M(v, 0.m, 1.m, m') = (m' = 1.m)$

Law 2.3.119. $(0.m, 1.m := v, v); M(m, 0.m, 1.m, m') = m' = v$

The following theorem provides us with the result that combined with theorem 2.3.113 ensures our parallel by merge operator preserves the design theory's healthiness conditions.

Theorem 2.3.120. *M preserves **H1** to **H4**.*

Finally, our definition of M makes it resemble the behaviour of a *selector*. By selector we mean a deterministic function that will select from a set of values according to the one that is different from a reference value. More formally:

Definition 2.3.121. *A deterministic selection function*

$$SELECT(m, 0.m, 1.m) =_{df} (m \lhd m = 1.m \rhd 1.m) \lhd m = 0.m \rhd 0.m$$

The fact that M behaves like a selector is established by the following theorem of a refinement relationship between M and an assignment using the selector function defined before.

Theorem 2.3.122. *Selection function – M equivalence*

$$M(m, 0.m, 1.m, m') \sqsubseteq m := \mathit{SELECT}(0.m, 1.m, m)$$

2.3.9 Synchronous, shared-variable parallelism

In this section we address the main limitation of the other forms of parallelism presented up to this point: that no process can ever reliably access the results produced by all other processes in the system. By definition, processes synchronise when they communicate with each other and agree on the value for the variables shared among them.

In the context of the UTP, each process can synchronise with its environment by means of committing its updated values for the shared variables for them to be merged. At the beginning of the next clock cycle, all synchronous processes update their local copies of the shared variables with the value calculated by the merging predicate.

It is clear that the additional behaviour introduced by programs that need to synchronise cannot be handled by the parallel by merge approach as described in the previous section. In the UTP [Hoare and He 1998, Chapter 7] this limitation is solved by means of incorporating three new variables: a clock-cycle counter c; and, for each shared variable m, the sequences $m.in$ and $m.out$. If m is the variable the process works on during the clock-cycle c, then $x.out_c$ is the value the process is committing to be merged (i.e., the value of m at the end of the clock cycle). On the other hand, $x.in_c$ is the merged value for m that all processes receive (and should set their local copies of m to) at the beginning of clock-cycle $c + 1$.

The relationship between m, $m.in$ and $m.out$ described above is maintained by means of the synchronisation operation **sync**. This operation is also responsible for keeping the clock-cycle counter set to the right value.

Definition 2.3.123. *The synchronisation operation*

$$\mathbf{sync} = (\mathbf{true} \vdash c, m, m.out_c := c + 1, m.in_c, m)$$

The merge operation from the previous section is used to calculate the value of the shared variables from the values in the local copies at the end of each clock-cycle. As outlined in the previous section, parallel by merge is defined as

$$(P \parallel_M Q) = ((P; U0(m)) \parallel (Q; U1(m))); M$$

hence the merging is performed once the parallel execution has terminated. In this context, the fact that P and Q may perform more than one clock-cycle makes the application of M insufficient to calculate the merged values at each of the synchronisation points. Furthermore, not only the value of m should be calculated by the merge, but also the merged values for c and the $m.out$ sequence.

What is necessary at this point is a new merge predicate that is capable of merging the whole behaviour of two programs. The new merge predicate should apply the M at all the synchronisation points that arose while the two processes were executed in isolation and return the merged behaviour that will result from their parallel execution.

Assuming parallel processes with the same duration (i.e., same number of clock-cycles), the merging of sequences of synchronisation actions is addressed by the final merge $\hat{M}_D$ predicate.

Definition 2.3.124. *Final merge*

$$\begin{aligned}\hat{M}_D \;=_{df}\; & (0.c = 1.c)_\perp ; c := 0.c \wedge \\ & M(m, 0.m, 1.m, m') \wedge \\ & \left\{ M((m \lhd i = c \rhd in_{i-1}), 0.out_i, 1.out_i, out'_i \mid c \le i < 0.c) \right\} \wedge \\ & \{ \mathit{II}_{\{out_i\}} \mid i < c \} ; \\ & \textbf{\textit{end}}\ 0.c, 1.c, 0.out, 1.out\end{aligned}$$

where $\{P_i \mid l \le i < n\}$ *stands for the parallel expression* $P_l \wedge P_{l+1} \wedge \cdots \wedge P_{n-1}$.

The first line of the above definition calculates the final value of the clock under the assumption that both branches had taken the same amount of clock cycles. The second line applies the merge predicate M to calculate the final value of m. The third line states that at each synchronisation point, the value of $m.out$ is computed on the assumption that m had the value input on the previous **sync** (if any). The fourth line ensures the values in the sequence at times before the current parallel process executed are kept constant. The final line finishes the scope of the variables introduced by the separating simulations.

The parallel-by-final-merge defined above satisfies Laws 2.3.106 to 2.3.111, like its asynchronous counterpart from the previous section. Furthermore, the abides principle gets specialised to incorporate synchronisation points.

Law 2.3.125. $(P; \textbf{\textit{sync}}; R) \parallel_{\hat{M}_D} (Q; \textbf{\textit{sync}}; S) = (P \parallel_M Q); \textbf{\textit{sync}}; (R \parallel_{\hat{M}_D} S)$

We have mentioned the $m.in$ sequence and we have described how the **sync** action uses it to receive the values that should be used after each process has synchronised. This operational understanding of the $m.in$ sequences can be easily specified in the relational calculus: the initial value of the $m.in$ sequence should be the same as the final (merged) value of the $m.out$ sequence. The definition of parallel by merge of synchronous processes with shared variables is as follows.

Definition 2.3.126. *Shared variables*

$$\begin{aligned}(\textbf{\textit{shared}}\ m; P) \;=_{df}\; & \textbf{\textit{var}}\ m.in, m.out, c; (m.out, c := \langle\rangle); \\ & (P \wedge (m.in = m.out')); \textbf{\textit{end}}\ m.in, m.out, c\end{aligned}$$

2.3.10 Normal forms

A collection of increasingly complex and general normal forms are presented in the UTP [Hoare and He 1998, Chapter 5]. Normal forms are defined to cope with assignment, non-determinism,

non-termination and recursion, with the goal of covering the complete set of programming constructs, and to address general aspects of the theory in a more precise way.

In this section we present only the normal forms dealing with assignment and recursion. We use the results from the assignment normal form in the transformation from first to second normal form in Section 5.2.3 while the results from the recursion-capable normal form are useful when dealing with recursion in the next chapter.

2.3.10.1 Assignment normal form

The first normal form we are interested in describing is the total assignment, in which all the program's variables appear on the left hand side in a certain order, and all expressions on the right hand side are well defined.

Definition 2.3.127. *Assignment normal form*

$$x, y, ..., z := e, f, ..., g$$

A partial assignment can be transformed into a total one by adding the identity assignment to the missing variables:

Law 2.3.128. $(x, y, \cdots := e, f, \dots) = (x, y, \dots, z := e, f, \dots, z)$

The list of variables in a total assignment can be permuted provided the expressions on the right-hand side of the assignment are subjected to the same permutation.

Law 2.3.129. $(x, \dots, y, z, \cdots := e, \dots, f, g, \dots) = (x, \dots, z, y, \cdots := e, \dots, g, f, \dots)$

As captured by Law 2.3.11, it is possible to substitute a variable by the value assigned to it in the precedent statement. Based on this result, the following law allows the elimination of sequential composition from the normal form.

Law 2.3.130. $(v := g; v := h(v)) = (v := h(g))$

Similarly, the following result is based in Law 2.3.26 to provide a mechanism to eliminate the conditional statement.

Law 2.3.131. $((v := g) \lhd b \rhd (v := h)) = (v := (g \lhd b \rhd h))$

2.3.10.2 Recursion-capable normal form

The introduction of recursion into the language permits the construction of a program whose degree of non-determinism cannot be expressed by finite means. The solution is to express the program as an infinite sequence of expressions

$$S = \{S_i \mid i \in \mathbb{N}\}$$

where each S_i is in finite normal form as defined in the previous section. In this way we can express all the possible behaviours of the program S together with some impossible ones as well.

To solve this problem, we request that each S_{i+1} is potentially stronger than its predecessor (i.e., $S_{i+1} \sqsupseteq S_i, \forall i \in \mathbb{N}$). The exact behaviour of the program is captured by the least upper bound of the whole sequence. This is the normal form for programs that contain recursive behaviour:

Definition 2.3.132. *Recursion-capable normal form*

$$\left(\bigsqcup_i S_i\right) \text{ or more briefly } \left(\bigsqcup S\right)$$

The distribution laws that allow us to eliminate the other operators of the language are as follows:

Law 2.3.133. $(\bigsqcup S) \sqcap P = \bigsqcup_i (S_i \sqcap P)$

Law 2.3.134. $P \lhd b \rhd (\bigsqcup S) = \bigsqcup_i (P \lhd b \rhd S_i)$

Law 2.3.135. $(\bigsqcup S); P = \bigsqcup_i (S_i; P)$

Law 2.3.136. *Provided P is a finite normal form, we have that:*

$$P; (\bigsqcup S) = \bigsqcup_i (P; S_i)$$

The following laws are required to eliminate the operators of the language when both of their arguments are in the normal form.

Law 2.3.137. $(\bigsqcup S) \lhd b \rhd (\bigsqcup T) = \bigsqcup_i (S_i \lhd b \rhd T_i)$

Law 2.3.138. $(\bigsqcup S); (\bigsqcup T) = \bigsqcup_i (S_i; T_i)$

The final law allows us to express recursion normal form, provided the recursive function preserves the order structure and least upper bounds (i.e., it is continuous).

Law 2.3.139. *Provided F is continuous we have that:*

$$\mu X \bullet F(X) = \bigsqcup_i F^i(\mathit{true})$$

Part II

Handel-C, its semantics and extensions for reasoning

Chapter 3

A theory of synchronous designs

"Notations are a frequent complaint... but the real problem
is to understand the meaning and properties of the symbols
and how they may and may not be manipulated, and to gain
fluency in using them to express new problems, solutions and proofs"

— C.A.R. Hoare

The goal of this chapter is to provide a semantic framework in which it is possible to reason about shared-variables, synchronous, parallel programs. As outlined in Chapter 1, our initial intention was to use the theory of designs from the UTP described in the previous chapter as the semantic domain for our reasoning framework. The main motivations for this choice were the fact that the design theory contains all the programming features described above together with a comprehensive set of laws that would be of great advantage in our context. However, while trying to prove some additional laws needed in our context we discovered a series of limitations that hindered the usability of the design theory as the semantic foundation for our reasoning framework. Section 3.1 explains these limitations in detail, yet they can be summarised as follows:

1. The final merge operator used to define the parallel by merge construct cannot handle programs of different length.

2. The semantics of assignment in Handel-C cannot simply be the assignment design in the UTP. Moreover, the semantics of the assignment construct in Handel-C needs to denote the advance in time (assignment takes one clock cycle) and be independent of the context in which it is placed (i.e., in sequential or parallel composition).

3. In the context of shared variables, it is not possible to assume $x = e$ right after assigning e to x provided x is a local variable. More formally, it does not hold in the design theory that:

$$(\mathbf{var}\ x; P; x := e; \mathbf{sync}; Q; \mathbf{end}\ x) = (\mathbf{var}\ x; P; x := e; \mathbf{sync}; (x = e)^{\top}; Q; \mathbf{end}\ x)$$

To address these limitations, we take the UTP approach and introduce additional observation variables and healthiness conditions to the theory of designs to define our own *synchronous theory*.

The new observation variables represent: (a) the clock-cycle counter (necessary to keep track of time and synchronisation points); and (b) clock-indexed sequences keeping the history of each variable in the program. The new healthiness conditions, in turn, ensure that: (i) the time always progresses forward (i.e., no "time travelling"); (ii) history is preserved (i.e., the past cannot be modified); and (iii) the synchronisation mechanism is controlled only by the theory and not by any of the individual processes.

As expected, the addition of new observation variables and healthiness conditions mentioned above restricts the theory of designs and creates a whole new theory. If we are to use a new theory as the reasoning language for our compiler we would normally need to provide definitions of all programming operators we are interested in and prove all the laws we will later on use for reasoning. Fortunately, we can take advantage of the "unifying" approach of the UTP and use the healthiness conditions as the link bridging the gap between the theory of designs and our new theory[1]. In this way, all basic constructs in the new theory are the "healthy" versions of their design counterparts and operators only need to be shown to be closed under the healthiness conditions. On the other hand, we still need to prove the validity of the algebraic laws for the reasoning language in the new theory. Luckily, a considerable number of this proofs follow from their design-based equivalent or can be proved using the same proof strategy. Section 3.2 describes the foundations of the theory of synchronous designs, its observations and healthiness conditions. In Section 3.3 we show that most of the results from Section 2.3 still hold in the new theory and provide additional laws that are needed in later chapters.

3.1 Limitations of the theory of design in our context

The first issue arising from trying to use the theory of designs to model programs in our context is the requirements imposed to achieve parallel behaviour: only processes that take the same amount of time can be put in parallel using the results from Section 2.3.9. We do not want to restrict ourselves to this type of parallel behaviour as most programming languages in our context do not pose any restrictions on how processes can be composed in parallel. In fact, it is common practice to have a never-terminating server process (like a driver for a hardware device) in parallel with other finite-length processes that act as its clients.

On the other hand, the fact that we intend to reason in a context where the time model is designed to mimic synchronous behaviour and that most variables are shared, suggests our semantics should take advantage of the synchronous, shared-variable parallel environment from Definition 2.3.126. In this context, the second shortcoming of using the theory of designs as our reasoning framework is that synchronisation is only meant to happen within parallel environments. Our context comprises languages where synchronisation permeates all constructs independently of them being executed in sequence or in parallel. Furthermore, our interest in the synchronisation action (denoted as **sync** in the theory of designs) is not only based on its functionalities as the means of keeping all processes having consistent copies of the shared variables. We are also interested in

[1]Technically, the new healthiness conditions together with the design healthiness conditions $\mathbf{H1} \circ \mathbf{H2}$ form a Galois connection between the lattice of designs and the lattice of predicates in our synchronous theory.

the capabilities of this construct as the main (and only) way of controlling the way in which time is recorded in our model (i.e., the clock cycle counter). For example, the semantics of $x \overset{:=}{}_{\mathsf{HC}} e$ requires not only that all other processes to be notified of the new value for x but also to signal that a whole clock cycle has passed while performing this update.

An obvious solution to this problem is to assume the scope of the shared variables and the synchronous environment encompasses the whole program. In this way, assignments of the form $x \overset{:=}{}_{\mathsf{HC}} e$ can always be described by the expression $(x := e; \mathbf{sync})$. The problem arising within the synchronous context is that law

$$(x \overset{:=}{}_{\mathsf{HC}} e) = (x \overset{:=}{}_{\mathsf{HC}} e) \fatsemi (x = e)^{\top} \tag{3.1}$$

no longer holds in the theory. This fact can be explained in the context of parallel programs. If we consider the program $((x \overset{:=}{}_{\mathsf{HC}} e_1) \parallel_{\mathsf{HC}} (x \overset{:=}{}_{\mathsf{HC}} e_2))$, it is not always true that it is equivalent to the program $((x \overset{:=}{}_{\mathsf{HC}} e_1 \fatsemi (x = e_1)^{\top}) \parallel_{\mathsf{HC}} (x \overset{:=}{}_{\mathsf{HC}} e_2))$. In particular, Law (3.1) should not hold as both assignments are trying to update x and, in general, it is not clear which value will be produced after the merge takes place.

On the other hand, if x is not a shared variable across parallel processes, there is no reason for this law to not to hold. From the semantics perspective, if we try to prove Law (3.1) we can follow the reasoning below:

$$\begin{aligned}
& (x \overset{:=}{}_{\mathsf{HC}} e) \\
= \; & \{\text{Proposed semantics for Handel-C assignment}\} \\
& x := e; \mathbf{sync} \\
= \; & \{\text{Definition 2.3.123 and law 2.3.11}\} \\
& c, x, x.out_c := c + 1, x.in_c, e
\end{aligned}$$

From this point it is necessary to bind $x.in_c$ with the value of e if we are to prove that $x = e$ after the assignment. If we recall from Definition 2.3.126, expressions in the shared context are subject to the additional condition that $x.in = x.out'$, hence if our program is of the form $(\mathbf{shared}\ x; (x \overset{:=}{}_{\mathsf{HC}} e); P)$ we can follow a reasoning path similar to the one presented above and reach a situation where

$$((c, x, x.out_c := c + 1, x.in_c, e); P) \wedge x.in = x.out'$$

If we were able to obtain $(c, x, x.out_c := c + 1, x.in_c, e) \wedge x.in = x.out'$ from the above equation, we could actually prove Law (3.1). In order to achieve that, we need to be able to show that

$$((P; Q) \wedge x.in = x.out') = (P \wedge x.in = x.out'); (Q \wedge x.in = x.out') \tag{3.2}$$

Unfortunately, the result described in equation 3.2 does not hold for the general case as it requires additional restrictions on c, $x.out$ and $x.in$ that are not met in the theory of designs.

An alternative semantic framework in our context is the UTP theory of reactive processes

[Hoare and He 1998, Chapter 8]. This theory is expressive enough to cover all our needs, as it was originally devised to cope with non-deterministic behaviour and blocking communications. In fact, it can be regarded as covering a superset of the required features for our context. On the other hand, it is the ability to cope with refusals and waiting states makes the reactive theory too detailed for us and, thus, unnecessarily complex to reason with in our synchronous yet deterministic context.

The semantic domain we are looking for is a new theory that lies in between the design and the reactive processes theories. We follow the same approach used in the UTP to further restrict the design theory to a new theory where the missing properties from this section can be shown to hold[2]. The next section presents our main contribution in this direction: a new theory of synchronous processes with shared variables. The new theory addresses all the issues mentioned in this section, and provides a comprehensive set of algebraic laws that enable us to reason about synchronous parallel programs with shared variables.

3.2 The theory of synchronous designs

The fact that we are dealing with shared-variables in a synchronous environment, indicates a solution based on the ideas presented in the shared environments from Definition 2.3.126. We first note that the key features of this definition are the clock counter c and the sequences $x.in$ and $x.out$. Our theory of synchronous programs includes these same variables, but with two main differences:

- *Global observational variables*. c, $x.in$ and $x.out$ are observational variables and no longer take part in the "implicit" lists of variables such as v in the definition of assignment. Like ok and ok', these variables have global scope that is implicitly opened and closed before and after the whole program respectively.

- *Implicit initialisation and binding*. The new variables are implicitly initialised ($c = 0$ and $x.out = \langle\rangle$) before the beginning of the program , while $x.in$ to $x.out'$ are implicitly bound by the expression $x.in = x.out'$ at the end of the program.

As with the design theory, we are not interested in all the predicates that can be expressed in the context of the syntax described above. In particular, we are only interested in programs that can advance the clock cycle counter forward, that do not modify the history of each variable x kept in the corresponding $x.out$ sequence and that keep $x.in$ constant as described by the healthiness conditions below:

Definition 3.2.1. *Core synchronous healthiness conditions*

S1 $P = P \wedge (c \leq c')$

S2 $P = P \wedge x.out \leq x.out'$

S3 $P = P \wedge x.in' = x.in$

[2]This same approach was used to solve the limitations of the alphabetised relational calculus when dealing with non-termination. In that case, the solution was to introduce the ok and ok' observations and to incorporate healthiness conditions, leading to the definition of the theory of designs.

Note that operator $\leq$ is overloaded in the above definition: it denotes the standard ordering for numbers (i.e., "less than or equal to" relation) in **S1**, yet it stands for "sequence prefix" in **S2**. On the other hand, it is common practice in the UTP to consider healthiness conditions not only as a predicate (i.e., "is P healthy?") but also as a function that "makes P healthy". In this context, we will use the notation **S** to refer to the simultaneous application of the three healthiness conditions described above:

Definition 3.2.2. *Synchronous healthiness condition*

$$\mathbf{S} = \mathbf{S1} \circ \mathbf{S2} \circ \mathbf{S3}$$

All of our healthiness conditions are conjunctive: they are functions from predicates to predicates defined by the form $H(P) = P \wedge \psi$ (i.e., P is a fixed point of H). A large number of healthiness conditions used to characterise UTP theories are defined by a conjunction of this form (e.g., the theory of reactive processes [Hoare and He 1998, Chapter 8] used for giving semantics to ACP [Bergstra and Klop 1985], CSP [Hoare 1983] and its extensions, like the semantics of *Circus* [Woodcock and Cavalcanti 2001] as described in [Woodcock and Cavalcanti 2002]).

In general, a number of properties are satisfied by conjunctive-healthy (**CH**) predicates, independently of the particular definition of ψ. Harwood, Woodcock and Cavalcanti [Harwood et al. 2008] explored, among other results, the closure of conjunctive-healthy predicates regarding the programming operators described in the previous sections. Some of the most relevant results of their work that are used in this thesis are presented below. We begin by establishing that the core UTP operators defined in the alphabetised relational calculus (Chapter 2) are closed within the subtheory delimited by conjunctive healthiness conditions. In particular, the sequential composition of **S**-healthy predicates is **S**-healthy.

Theorem 3.2.3. *Sequential composition*

$$[\mathbf{S}(P) \wedge \mathbf{S}(Q) \Rightarrow \mathbf{S}(P; Q)]$$

Furthermore, the conjunction, disjunction and selection of **S**-healthy predicates are also **S**-healthy.

Theorem 3.2.4. *Conjunction, disjunction and selection closure*

$$[\mathbf{S}(P) \wedge \mathbf{S}(Q) \Rightarrow \mathbf{S}(P \wedge Q) \wedge \mathbf{S}(P \vee Q) \wedge \mathbf{S}(P \lhd b \rhd Q)]$$

Provided the body of a recursive program is **S**-healthy, then the whole recursive program is **S**-healthy.

Theorem 3.2.5. *Recursive* **S***-healthy programs*

$$[\mathbf{S}(P) \Rightarrow \mu X \bullet (P; X) \lhd b \rhd \textit{II}]$$

It is important to notice that any theory of conjunctive-healthy predicates is disjoint from the theory of designs: on abortion, a design provides no guarantees while a conjunctive-healthy pre-

dicate requires ψ to hold. Furthermore, it is possible to link the theory of designs with a theory of **CH**-healthy predicates by taking **CH** as the *approximate relationship* between designs and **CH**-healthy predicates. In the context of **S**-healthy predicates, we can instantiate this general result to the following theorem:

Theorem 3.2.6. *Galois connection with the theory of designs*

$$P \sqsubseteq \mathit{H1} \circ \mathit{H2}(Q) \quad \textit{if and only if} \quad \mathbf{S}(P) \sqsubseteq Q$$

where P is a design and Q is **S***-healthy*

A direct consequence of this result is that predicates in the design theory can be used to find their approximated counterparts in the synchronous designs lattice. For example, the skip programming construct in the synchronous theory is just II_D made **S**-healthy:

Definition 3.2.7. *Synchronous skip*

$$\mathit{II} =_{df} \mathbf{S}(\mathit{II}_D)$$

In addition to the general results that can be derived from **S** being a **CH** healthiness condition, our formulation also satisfies other properties that proved very useful later on in this chapter.

Theorem 3.2.8. $\mathit{II} \Rightarrow \mathbf{S}(\mathit{true})$

It is also straightforward to conclude that II is **S**-healthy.

Theorem 3.2.9. $\mathbf{S}(\mathit{II}) = \mathit{II}$

The healthiness conditions are transitive when combined by means of the sequential composition operator.

Theorem 3.2.10. **S** *is transitive*

$$\mathbf{S}(\mathit{true}); \mathbf{S}(\mathit{true}) = \mathbf{S}(\mathit{true})$$

The healthiness conditions preserve the refinement ordering: they are monotonic regarding $\sqsubseteq$.

Theorem 3.2.11. **S** *is monotonic*

$$(P \sqsubseteq Q) \Rightarrow (\mathbf{S}(P) \sqsubseteq \mathbf{S}(Q))$$

provided P and Q are UTP designs.

S distributes over conjunction and disjunction.

Theorem 3.2.12. **S** *distributes over conjunction*

$$\mathbf{S}(P \wedge Q) = \mathbf{S}(P) \wedge \mathbf{S}(Q)$$

Theorem 3.2.13. **S** *distributes over disjunction*

$$\mathbf{S}(P \vee Q) = \mathbf{S}(P) \vee \mathbf{S}(Q)$$

If we have the conjunction of an **S**-healthy predicate with another predicate, then the whole predicate is **S**-healthy.

Theorem 3.2.14. **S** *extends over conjunction*

$$P \wedge \mathbf{S}(Q) = \mathbf{S}(P \wedge Q)$$

3.2.1 H3 in the synchronous theory

As mentioned in Section 2.3, **H3** is satisfied only by those designs where the precondition does not mention dashed variables (i.e., the precondition is a condition instead of a more general predicate). Algebraically, **H3**-healthy predicates have the design skip as their right unit for sequential composition.

In the context of **S**-healthy designs, the conjunctive healthiness conditions mentioning dashed variables makes it impossible to eliminate all after variables from the preconditions. On the other hand, the healthiness conditions only restrict the values of c', $x.in'$ and $x.out'$. Thus, it should be possible to formulate a notion similar to **H3** provided the observational variables in the precondition can be "ignored". As in the case of **H3**, the notion can be precisely captured by having the theory's skip construct as right unit for sequential composition. More formally, the notion of **H3** has its equivalent notion in the lattice of **S**-healthy designs by means of **SH3** defined as follows.

Definition 3.2.15. *Lifted **H3***

$$\mathbf{SH3} \;\; P = P; \mathit{II}$$

provided P is **S***-healthy.*

As expressed before, the precondition of **SH3**-healthy designs does not mention dashed variables other than c', $x.in'$ and $x.out'$. The following theorem captures this notion more precisely.

Theorem 3.2.16.

An **S** *healthy design* $\mathbf{S}(P_1 \vdash P_2)$ *is* **SH3** *if and only if its precondition does not mention dashed variables other than the observations c, x.out and x.in.*

As with the design theory, we need to show that **SH3** is closed under the basic operators of the language (i.e., sequential composition, selection and recursion). Regarding sequential composition, provided P and Q are **SH3**-healthy, then $P; Q$ is **SH3**-healthy.

Theorem 3.2.17. **SH3** *sequential composition closure*

$$P; Q = \mathbf{SH3}(P; Q)$$

provided P and Q are **SH3***-healthy.*

Similarly, provided P and Q are **SH3**-healthy, the selection of either of them is also **SH3**-healthy.

Theorem 3.2.18. SH3 *conditional closure*

$$P \lhd b \rhd Q = \mathbf{SH3}(P \lhd b \rhd Q)$$

provided P and Q are **SH3***-healthy.*

To show that recursion is also closed under **SH3**, we take the same approach used by Hoare and He [1998, Chapter 3.1] to show that the least fixed point of a design is also a design. The programming operators defined so far are monotonic and closed within the **SH3** lattice. The fact that all operators map **SH3**-healthy designs to **SH3**-healthy designs and that recursion is solely built up from constructs with this property, is enough to show that recursion preserves **SH3**.

3.3 Recasting the design theory

In this section we revise the definitions and algebraic properties of all operators we introduced in chapter 2 in the light of our conjunctive healthiness conditions and the synchronous context. We also take the chance to explore additional algebraic properties of some of the operators as they are needed in later chapters where the compilation process is addressed.

We also provide new definitions for alphabet extension and parallel composition. These are the only definitions we needed to re-formulate. All other definitions only needed to be "lifted" from the design theory to satisfy the healthiness conditions. The new definition of alphabet extension is necessary to denote the effect a program has on the trace $x.out$ when forced to keep a fresh variable x constant. As the design theory did not keep track of the variable histories it did not need to take this detail into account.

The need for a different treatment of parallel composition lies in the fact that there is no basic definition for it in the alphabetised propositional calculus. Instead, a definition was postulated in the theory of designs. In our context, the definition needs to be slightly modified in order to account for the fact that the predicates being composed in parallel need to be **S**-healthy.

3.3.1 Miracle, abort and refinement in the synchronous designs theory

As mentioned before, **S** can be seen as an approximate relationship between designs and their **S**-healthy counterparts. From this observation, it is easy to see that we can obtain the **S**-healthy equivalent of the significant elements (like the top and bottom of the lattice) by means of calculating their image through **S**. The presence of conjunctive healthiness conditions require the predicates to remain *healthy* even when they cannot be started or their precondition is not satisfied. Furthermore, designs are required to satisfy the healthiness condition even when they diverge.

In the case of the top of the design lattice, it is the design that performs miracles and it cannot be started, yet it needs to maintain the healthiness conditions.

Definition 3.3.1. *Synchronous miracle*

$$\top =_{df} \mathbf{S}(\top_D)$$

The bottom of the synchronous lattice is the diverging design with the restriction that unhealthy behaviour is not allowed.

Definition 3.3.2. *Synchronous abort*

$$\bot =_{df} \mathbf{S}(\bot_D)$$

Refinement follows the same notion shared by all UTP theories: the behaviour of an implementation should imply its specification closed under their universally quantified alphabet. Moreover, the monotonicity of the healthiness condition regarding refinement gives the same intuitive view of refinements in the theory of designs: we introduce refinement by either weakening the precondition, or strengthening the postcondition in the presence of the precondition. This is established by the result below.

Theorem 3.3.3. *Synchronous designs refinement*

$$\mathbf{S}(Q_1 \vdash Q_2) \sqsubseteq \mathbf{S}(P_1 \vdash P_2) \quad \textit{if and only if} \quad [Q_1 \Rightarrow P_1] \wedge [(Q_1 \wedge P_2) \Rightarrow Q_2]$$

All predicates in the synchronous theory refine the synchronous abort and are, in turn, refined by miracle.

Law 3.3.4. $\top \sqsupseteq \mathbf{S}(P)$

Law 3.3.5. $\bot \sqsubseteq \mathbf{S}(P)$

3.3.2 Sequential composition

We have already shown that sequential composition is closed under **S** and **SH3**-healthy predicates. Moreover, we have shown that if P and Q are $\mathbf{S} \circ \mathbf{H1} \circ \mathbf{H2}$-healthy (i.e., they are **S**-healthy designs), the result is also $\mathbf{S} \circ \mathbf{H1} \circ \mathbf{H2}$-healthy. In this section we focus on the algebraic properties that characterise the sequential composition of synchronous designs.

The combination of two **S**-healthy designs is a new **S**-healthy design of the form described below.

Theorem 3.3.6. *Sequential composition of* **S***-healthy designs*

$$\mathbf{S}(P_1 \vdash P_2); \mathbf{S}(Q_1 \vdash Q_2) = \mathbf{S}(\neg(\mathbf{S}(\neg P_1); \mathbf{S}(\textit{true})) \wedge \neg(\mathbf{S}(P_2); \mathbf{S}(\neg Q_1)) \vdash \mathbf{S}(P_2); \mathbf{S}(Q_2))$$

This result states that provided the healthy precondition P_1 can be established (this is denoted by the expression $\neg(\mathbf{S}(\neg P_1); \mathbf{S}(\mathbf{true}))$ above) and that the **S**-healthy execution of P_2 does not lead to a state where the precondition Q_1 cannot be satisfied ($\neg(\mathbf{S}(P_2); \mathbf{S}(\neg Q_1))$), the sequential composition of **S**-healthy designs behaves like the sequential composition $\mathbf{S}(P_2); \mathbf{S}(Q_2)$. A significant

consequence of theorem 3.2.16, is that provided P and Q are **SH3**, we can simplify the above result by replacing $\neg(\mathbf{S}(\neg P_1); \mathbf{S}(\mathbf{true}))$ simply by $\neg\mathbf{S}(\neg P_1)$.

Algebraically speaking, to precede a program P by the miraculous program results altogether in a miracle, thus, $\top$ is a left zero for sequential composition.

Law 3.3.7. $\top; \mathbf{S}(P) = \top$

As with designs, a program P cannot recover from divergence. More precisely, $\bot$ is a left zero for sequential composition.

Law 3.3.8. $\bot; \mathbf{S}(P) = \bot$

Sequential composition of **S**-healthy designs has II as left unit and, provided the predicate on the left is **SH3**, also as its right unit.

Law 3.3.9. $\mathit{II}; \mathbf{S}(P_1 \vdash P_2) = \mathbf{S}(P_1 \vdash P_2)$

Law 3.3.10. *Provided P is* **SH3** *we have:*

$$\mathbf{S}(P); \mathit{II} = \mathbf{S}(P)$$

Sequential composition of **S**-healthy designs distributes through the least upper bound operator.

Law 3.3.11. $(\bigsqcup S); P = \bigsqcup_i (S_i; P)$

Law 3.3.12. *Provided P is in finite normal form, we have:*

$$P; (\bigsqcup S) = \bigsqcup_i (P; S_i)$$

3.3.3 Assignment

As with the skip construct, our definition of assignment in the synchronous theory is the image of assignment in the design theory through the healthiness condition **S**.

Definition 3.3.13. *Synchronous assignment*

$$x \overset{:=}{_{snc}} e \ =_{df} \ \mathbf{S}(x := e)$$

An interesting observation (and useful result in many proofs) is that, when the observations c, $x.out$ and $x.in$ are not explicitly modified by an assignment, the healthiness condition is absorbed by the propositional assignment. This observation follows from the fact that this kind of assignment keep the clock-cycle counter and the history variables constant, trivially satisfying the healthiness conditions.

Theorem 3.3.14. *Provided x is not an observation variable we have that:*

$$\mathbf{S}(x := e) = x := e$$

A direct consequence of Definition 3.3.13 and theorem 3.3.14 is that the synchronous assignment construct is **SH3** as described by the following theorem.

Theorem 3.3.15. *The assignment construct is* **SH3**

A sequence of assignments to the same variable can be simplified to a single assignment.

Law 3.3.16. $(x :=_{snc} e; x :=_{snc} f(x)) = x :=_{snc} f(e)$

A simultaneous assignment can be performed by interleaving its individual assignments, provided certain conditions are met.

Law 3.3.17. *If $x \neq y$ and e_2 does not mention x we have that:*

$$(x :=_{snc} e_1; y :=_{snc} e_2) = (x, y :=_{snc} e_1, e_2)$$

The order of the list of variables in a multiple assignment is irrelevant and can be permuted, provided the list of expressions in the right hand side is permuted in the same way.

Law 3.3.18. $(x, y :=_{snc} e_1, e_2) = (y, x :=_{snc} e_2, e_1)$

A sequence of independent assignments can be executed in any order provided the expressions used in the assignments do not depend on the variables being updated.

Law 3.3.19. *Provided $x \neq y$, e_1 does not depend on y and e_2 does not depend on x then:*

$$(x :=_{snc} e_1; y :=_{snc} e_2) = (y :=_{snc} e_2; x :=_{snc} e_1)$$

3.3.4 Conditional

An unfortunate consequence of the way in which sequential composition operates is that conjunctive healthiness conditions (including **S**) do not distribute through it. This means that, in general, it is not possible to show that $\mathbf{S}(P; Q) \Rightarrow \mathbf{S}(P) \wedge \mathbf{S}(Q)$ (the other direction of this implication does hold, as shown by theorem 3.2.3). Conditional, on the other hand, is defined in terms of the disjunction connective. This fact enables the conjunctive healthiness conditions to distribute through the selection operator, as shown by the following lemma.

Theorem 3.3.20. *Selection of* **S** *designs*

$$\mathbf{S}(P) \lhd b \rhd \mathbf{S}(Q) = \mathbf{S}(P \lhd b \rhd Q)$$

We have already established (see Section 3.2) that the selection between two **S**-healthy designs is itself **S**-healthy. The following theorem provides us with its characterisation as an **S**-healthy design.

Theorem 3.3.21. *Design characterisation of* **S** *selection*

$$\mathbf{S}(P_1 \vdash P_2) \lhd b \rhd \mathbf{S}(Q_1 \vdash Q_2) = \mathbf{S}(P_1 \lhd b \rhd Q_1 \vdash P_2 \lhd b \rhd Q_2)$$

Provided P, Q and R are **S**-healthy, the selection of **S**-healthy designs satisfies the basic properties about the conditional showed in Section 2.3.3.

Law 3.3.22. $P \lhd b \rhd Q = Q \lhd \neg b \rhd P$

Law 3.3.23. $P \lhd b \rhd P = P$

Law 3.3.24. $P \lhd \mathit{true} \rhd Q = P$

Law 3.3.25. $P \lhd \mathit{false} \rhd Q = Q$

Law 3.3.26. $(P \lhd b \rhd Q) \lhd b \rhd R = P \lhd b \rhd R = P \lhd b \rhd (Q \lhd b \rhd R)$

Law 3.3.27. $(P \lhd b \rhd Q) \lhd c \rhd R = (P \lhd b \wedge c \rhd (Q \lhd c \rhd R))$

Law 3.3.28. $x \overset{:=}{_{snc}} e_1 \lhd b \rhd x \overset{:=}{_{snc}} e_2 = x \overset{:=}{_{snc}} (e_1 \lhd b \rhd e_2)$

Law 3.3.29. $(P \lhd c \rhd Q) \lhd b \rhd R = (P \lhd b \rhd R) \lhd c \rhd (Q \lhd b \rhd R)$

Law 3.3.30. $P \lhd b \rhd (P \lhd c \rhd Q) = P \lhd (b \vee c) \rhd Q$

Law 3.3.31. $\neg(s \wedge b) \wedge (P \lhd (s \wedge \neg b) \rhd Q) = \neg(s \wedge b) \wedge (P \lhd s \rhd Q)$

Law 3.3.32. $x \overset{:=}{_{snc}} e; (P \lhd b(x) \rhd Q) = (x \overset{:=}{_{snc}} e; P) \lhd b(e) \rhd (x \overset{:=}{_{snc}} e; Q)$

Law 3.3.33. $(P \lhd b \rhd Q); R = (P; R) \lhd b \rhd (Q; R)$

3.3.5 Assertion and assumption

The notions in this section are similar to the ones defined in Section 2.3.5 but with the results lifted to the synchronous design theory. The following two results capture this notion in the definitions of the synchronous assumption and assertion constructs.

Definition 3.3.34. *Assumption in the synchronous environment*

$$b^{\top}_{S} = \mathit{II} \lhd b \rhd \top$$

Definition 3.3.35. *Assertion in the synchronous environment*

$$b^{\top}_{S} = \mathit{II} \lhd b \rhd \bot$$

A reassuring result is that we can obtain the same definitions if we take the design view of an assertion/assumption and make it **S**-healthy.

Theorem 3.3.36. *Design assumption – synchronous assumption equivalence*

$$b^{\top}_{S} = \mathbf{S}(b^{\top})$$

Theorem 3.3.37. *Design assertion – synchronous assertion equivalence*

$$b^{\top}_{S} = \mathbf{S}(b^{\top})$$

Assuming (asserting) a false condition leads to miracle (abort).

Law 3.3.38. $\mathit{false}^{\top}_{S} = \top$

Law 3.3.39. $\mathit{false}^{\perp}_{S} = \perp$

The assertion of a condition b is refined by II (the assertion may result in abort if b does not hold, while II always terminates). II is, in turn, refined by assuming a condition b.

Law 3.3.40. $b^{\top}_{S} \sqsupseteq \mathit{II} \sqsupseteq b^{\perp}_{S}$

The following laws allow the simplification of sequences of assertions and assumptions.

Law 3.3.41. $b^{\top}_{S}; b^{\perp}_{S} = b^{\top}_{S}$

Law 3.3.42. $b^{\perp}_{S}; b^{\top}_{S} = b^{\perp}_{S}$

Law 3.3.43. $b^{\top}_{S}; b^{\perp}_{S} \sqsupseteq \mathit{II}$

Law 3.3.44. $b^{\perp}_{S}; b^{\top}_{S} \sqsubseteq \mathit{II}$

Law 3.3.45. $b^{\top}_{S}; c^{\top}_{S} = (b \wedge c)^{\top}_{S}$

Law 3.3.46. $b^{\top}_{S}; c^{\top}_{S} = c^{\top}_{S}; b^{\top}_{S}$

Law 3.3.47. $b^{\perp}_{S}; c^{\perp}_{S} = (b \wedge c)^{\perp}_{S}$

Law 3.3.48. $(b \vee c)^{\top}_{S}; b^{\top}_{S} = b^{\top}_{S}$

Law 3.3.49. $b^{\perp}_{S}; b^{\perp}_{S} = b^{\perp}_{S}$

Law 3.3.50. $b^{\top}_{S}; b^{\top}_{S} = b^{\top}_{S}$

The assumption and assertion constructs are **SH3**-healthy.

Law 3.3.51. $b^{\top}_{S}; \mathit{II} = b^{\top}_{S}$

Law 3.3.52. $b^{\perp}_{S}; \mathit{II} = b^{\perp}_{S}$

It is always possible to assume or assert the condition $x = e$ after an assignment updating x to the value e (provided e does not mention x).

Law 3.3.53. *Provided e_1 does not mention x or y we have:*

$$(x, y \overset{:=}{_{snc}} e_1, e_2) = (x, y \overset{:=}{_{snc}} e_1, e_2); (x = e_1)^{\top}_{S}$$

Law 3.3.54. *Provided e_1 does not mention x or y we have:*

$$(x, y \overset{:=}{_{snc}} e_1, e_2) = (x, y \overset{:=}{_{snc}} e_1, e_2); (x = e_1)^{\perp}_{S}$$

Two variables that have been made equal by an assignment can be exchanged when used in a condition inside an assertion.

Law 3.3.55. $(x \mathrel{:=}_{snc} y; (y = e)^{\perp}_{S}) = (x \mathrel{:=}_{snc} y; (x = e)^{\perp}_{S})$

If the variables in the assignment are not mentioned in an assumption, the order between them can be swapped.

Law 3.3.56. *Provided b does not depend on x we have that:*

$$x \mathrel{:=}_{snc} e; b^{\top}_{S} = b^{\top}_{S}; x \mathrel{:=}_{snc} e$$

When evaluating a conditional, it is possible to assume the value of the condition inside each of its branches.

Law 3.3.57. $P \lhd b \rhd Q = (b^{\top}_{S}; P) \lhd b \rhd ((\neg b)^{\top}_{S}; Q)$

The following laws allow the simplification of the conditional construct based on knowledge provided by assertions or assumptions.

Law 3.3.58. $b^{\top}_{S}; (P \lhd b \rhd Q) = b^{\top}_{S}; P$

Law 3.3.59. $(\neg b)^{\top}_{S}; (P \lhd b \rhd Q) = (\neg b)^{\top}_{S}; Q$

Law 3.3.60. *Provided* $\neg(b \wedge c)$ *we have:*

$$b^{\top}_{S}; (P \lhd c \rhd Q) = (b^{\top}_{S}; Q)$$

Law 3.3.61. *Provided P and Q are* **S***-healthy we have:*

$$c^{\top}_{S}; (P \lhd b \wedge c \rhd Q) = c^{\top}_{S}; (P \lhd b \rhd Q)$$

Assumptions and assertions distribute rightwards with respect to the conditional construct.

Law 3.3.62. $b^{\top}_{S}; (P \lhd c \rhd Q) = (b^{\top}_{S}; P) \lhd c \rhd (b^{\top}_{S}; Q)$

Law 3.3.63. $b^{\perp}_{S}; (P \lhd c \rhd Q) = (b^{\perp}_{S}; P) \lhd c \rhd (b^{\perp}_{S}; Q)$

3.3.6 Dynamic scope

Being able to reason about variables and their scope will be crucial in later chapters when the compilation of Handel-C programs is addressed. As the whole reasoning framework is based on synchronous designs, we need to provide means of controlling variable blocks in this theory. A remarkable fact of the UTP and how its theories relate to each other is the fact that we can define the scope delimiters in the synchronous theory by means of the same constructs introduced in the alphabetised relational calculus but with the additional restriction that the healthiness conditions must also hold.

Definition 3.3.64. *Start scope*

$$\mathtt{var}\ x\ =_{df}\ \mathbf{S}(\textbf{\textit{var}}\ x)$$

Definition 3.3.65. *End scope*

$$\mathtt{end}\ x\ =_{df}\ \mathbf{S}(\textbf{\textit{end}}\ x)$$

The following theorems show that var and end are, as in the design and predicate calculus theories, nothing more than existential quantification.

Theorem 3.3.66. *Provided P is* **S***-healthy and x is not an observational variable we have:*

$$\mathtt{var}\ x; P = \exists x \bullet P$$

Theorem 3.3.67. *Provided P is* **SH3***-healthy and P does not mention x', and x is not an observational variable we have:*

$$P; \mathtt{end}\ x = \exists x' \bullet P$$

Note that an interesting consequence of the result above is that for any **S**-healthy predicate P, we can replace P; end x with its the propositional end of scope for x: P; **end** x. This is allowing us to directly use the design dynamic scope delimiters in our synchronous theory, however, we decided to keep our new notation to syntactically reinforce the fact that all predicates are **S**-healthy. On the other hand, the propositional constructs to open and close the scope of a variable absorb the healthiness conditions.

Theorem 3.3.68. $\mathbf{S}(\textbf{\textit{var}}\ x) = \textbf{\textit{var}}\ x$

Theorem 3.3.69. $\mathbf{S}(\textbf{\textit{end}}\ x) = \textbf{\textit{end}}\ x$

All basic laws about the scope delimiters we introduced in Section 2.3.4 still hold for their synchronous counterparts. We recast them here for clarity and in order to be able to reference them in later sections.

Law 3.3.70. $\mathtt{var}\ x; \mathtt{end}\ x = \mathit{II}$

Law 3.3.71. $\mathtt{end}\ x; \mathtt{var}\ x \sqsubseteq \mathit{II}$

Law 3.3.72. $x \overset{:=}{_{snc}} e; \mathtt{end}\ x = \mathtt{end}\ x$

Law 3.3.73. $\mathtt{var}\ x \sqsubseteq (\mathtt{var}\ x; x \overset{:=}{_{snc}} e)$

Law 3.3.74. $\mathtt{var}\ x; (x = e)^{\top}_{S} = (\mathtt{var}\ x; x \overset{:=}{_{snc}} e)$

Law 3.3.75. $\mathtt{var}\ x; \mathtt{var}\ y = \mathtt{var}\ x, y$

Law 3.3.76. $\mathtt{end}\ x; \mathtt{end}\ y = \mathtt{end}\ x, y$

Law 3.3.77. $\mathtt{var}\, x, y = \mathtt{var}\, y, x$

Law 3.3.78. $\mathtt{end}\, x, y = \mathtt{end}\, y, x$

Law 3.3.79. *If x is not free in e then*

$$(\mathtt{end}\, x; \mathtt{var}\, x \mathrel{:=}_{snc} e) = (x \mathrel{:=}_{snc} e)$$

Law 3.3.80. *Provided P is* **SH3** *and neither e nor* **S** *mention x we have:*

$$(P; x \mathrel{:=}_{snc} e) = ((P; \mathtt{end}\, x)_{+x}; x \mathrel{:=}_{snc} e)$$

The next law enables us to expand the beginning of scope of a variable local to both branches of a conditional.

Law 3.3.81. *Provided b does not mention x we have:*

$$\mathtt{var}\, x; (P \lhd b \rhd Q) = (\mathtt{var}\, x; P) \lhd b \rhd (\mathtt{var}\, x; Q)$$

Up to this point we have presented an algebraic way of controlling the scope of variables in the particular case of the selection construct. In order to be able to present the laws that allow the manipulation of variable scopes over sequential composition, we need a mechanism to extend the alphabet of a given predicate P in our synchronous theory.

In Section 2.3.4 we showed how to extend the alphabet of a design $(P_1 \vdash P_2)$ to include a new variable x by means of making its postcondition keep the value of x constant (i.e., $(P_1 \vdash P_2)_{+x} = (P_1 \vdash P_2 \wedge x' = x)$). Unfortunately, this way of extending the alphabet is not enough in the theory of synchronous designs as it does not account for the advance of time and the effect it has on x's history. To see why, let's consider the case of the simple program

$$y \mathrel{:=}_{\mathrm{snc}} e_1; \mathtt{sync} \tag{3.3}$$

where x is not in the alphabet. If we expand the definition of `sync` (3.3.124), apply Law 3.3.16 and our definition of alphabet extension from the previous chapter we obtain:

$$\mathbf{S}(\mathbf{true} \vdash c, y, y.out_c := c + 1, y.in_c, e_1 \wedge x', x.out' = x, x.out)$$

The problem with this result is that neither x nor $x.out$ are being updated in the right way in order for synchronisation to take place. A correct alphabet extension that takes into account the effects of synchronisation should extend the alphabet of equation (3.3) to obtain:

$$\mathbf{S}(\mathbf{true} \vdash c, y, y.out_c, x, x.out_c := c + 1, y.in_c, e_1, x.in_c, x)$$

that is, $x' = x$ has to be lifted in the synchronous theory to also establish that x has been kept constant during all clock cycles the process with extended alphabet has been executing. The following definition describes our new alphabet extension operator that achieves this goal.

Definition 3.3.82. *Alphabet extension*

$$\mathbf{S}(P_1 \vdash P_2)_{[x]} =_{df} \mathbf{S}(P_1 \vdash P_2 \wedge E(x, c'))$$

where $E(x, c')$ is defined as follows:

Definition 3.3.83. *Alphabet extension predicate*

$$E(x, c') =_{df} (x', x.out'_c, x.in' = ((x.in_{c'-1}, x) \lhd c < c' \rhd (x, x.out_c)), x.in) \wedge \left\{x.out'_i = x.in_{i-1} \mid c < i < c'\right\} \wedge \{I\!I_{\{x.out_i\}} \mid i < c\}$$

Definition 3.3.83 captures the core notion of our alphabet extension mechanism. When the process of which the alphabet is being extended does not advance the clock cycle counter, alphabet extension behaves like the design's alphabet extension regarding x but it also states the history in $x.out$ should remain constant (after all, there is nothing to be added to it). On the other hand, if the process advances the clock cycle, x and $x.out$ are forced to take the values set to them by the synchronisation mechanism (remember the `sync` action within P does not synchronise on x). Finally, in both cases $x.out_{c+1} \ldots x.out_{c'-1}$ are set to the corresponding input value for x in the clock cycle.

In more general terms, the effects of alphabet extension can be captured in the following **S**-healthy design:

Definition 3.3.84.

$$E_D(x, x', c, c') =_{df} \mathbf{S}(\textit{true} \vdash (x', x.out'_c, x.in' = (x.in_{c'-1}, x) \lhd c < c' \rhd (x, x.out_c), x.in) \wedge \left\{x.out'_i = x.in_{i-1} \mid c < i < c'\right\} \wedge \{I\!I_{\{x.out_i\}} \mid i < c\})$$

It is be useful to capture the notion of alphabet extension as the sequential composition of two **S**-healthy designs. The main difficulty is that the definition of alphabet extension for a process P requires access to both the initial and final clock-cycle count of P (i.e., to the contents of the c and c' variables). Because of the way in which sequential composition is defined, it is easy to allow P's subsequent predicate to have access to P's final clock-cycle count. There is no way, however, to have access to P's initial value of c, as it gets hidden in the sequential composition. The solution is to introduce a new variable $0.c$ to store the final value of c while executing P and, after that, to make the value of c' equal to c. Provided the value of c at the beginning of P is c_0, and if we follow the modified P by a predicate Q, then the sequential composition definition produces $Q[c_0, 0.c/c, c']$ as we wanted. We use the outlined mechanism described before in the following definition of alphabet extension in the synchronous design theory.

Law 3.3.85. *Provided $P = (P_1 \vdash P_2)$ is a* **SH3**, **S***-healthy design we have:*

$$\mathbf{S}(P)_{[x]} = \mathbf{S}(P_1 \vdash P_2; \textbf{\textit{var}}\ 0.c := c; \textbf{\textit{end}}\ c)_{+x,c}; E_S(x)$$

where $E_S(x)$ is defined as follows:

Definition 3.3.86. *Design alphabet extension*

$$E_S(x) =_{df} \mathbf{S}(\textit{true} \vdash (E(x, 0.c) \wedge c := 0.c); \textit{end}\ 0.c)$$

The alphabet extension of a process that does not advance the clock cycle-counter is nothing but alphabet extension in the design theory.

Law 3.3.87. *Provided P is* **S***-healthy and that $P = P \wedge c' = c$ we have:*

$$P_{[x]} = P_{+x}$$

A straightforward consequence is that our new synchronous alphabet extension reduces to the design's alphabet extension when applied to $\mathbb{I}$

Law 3.3.88. $\mathbb{I}_{[x]} = \mathbb{I}_{+x} = \mathbb{I}$

Alphabet extension distributes over conditional and it leads to refinement when distributed over sequential composition.

Law 3.3.89. $(P \lhd b \rhd Q)_{[x]} = (P_{[x]} \lhd b \rhd Q_{[x]})$

Law 3.3.90. *Provided P and Q are* **S***-healthy we have:*

$$(P; Q)_{[x]} \sqsubseteq (P_{[x]}; Q_{[x]})$$

Pushing the alphabet extension operation into a loop's body also leads to refinement.

Law 3.3.91. *Provided P is* **SH3***-healthy we have:*

$$(b * P)_{[x]} \sqsubseteq b * P_{[x]}$$

As with the design theory, the scope of a variable can be extended over a **S**-healthy design. If the design does not synchronise with its environment (and, hence, it does not advance the clock-cycle counter), the expansion of scope operates in the same way it does for designs.

Law 3.3.92. *If P is* **SH3** *and* **S***-healthy, it does not perform* `sync` *events and neither P nor* **S** *mention x then we have that:*

$$P; \texttt{var}\ x; Q = \texttt{var}\ x; P_{[x]}; Q$$

Law 3.3.93. *If P and Q are* **SH3***-healthy, P does not perform any* `sync` *events and neither P nor* **S** *mention x' then we have that:*

$$Q; \texttt{end}\ x; P = Q; P_{[x]}; \texttt{end}\ x$$

On the other hand, if the process P we are trying to incorporate in the scope of a variable x does synchronise, then the expansion of the scope of x over P leads to refinement. The reason

for the introduction of refinement is the restriction alphabet extension places on the trace $x.out$. Before the alphabet was extended to include x in P, the only restriction on $x.out$ was imposed by **S2**[3]: $x.out \leq x.out'$. After extending the alphabet, P not only has to satisfy **S2**, but also imposes the restriction that $x.out$ has to be extended as if all the synchronisation actions would have had updated it as well.

Law 3.3.94. *If P is* **SH3***-healthy, it does perform at least one* `sync` *event and neither P nor* **S** *mention x then we have that:*

$$P; \texttt{var}\ x; Q \sqsubseteq \texttt{var}\ x; P_{[x]}; Q$$

Law 3.3.95. *If P and Q are* **SH3***-healthy, P does perform at least one* `sync` *event and neither P nor* **S** *mention x' then we have that:*

$$\texttt{end}\ x; P \sqsubseteq P_{[x]}; \texttt{end}\ x$$

3.3.7 Iteration

Provided P is **S**-healthy, the iteration construct $(b*P)$ in the synchronous design theory is defined as the least fixed point of the equation: $X = (P; X) \lhd b \rhd \text{II}$. Note that as with sequential composition and selection, the least fixed point operator remains the same as the one used in all other theories. After all, the least fixed point is an operator over boolean predicates describing programs and the programs in the synchronous theory are nothing more than particularly restricted predicates of the same kind. More formally:

Definition 3.3.96. *Iteration*

$$b * P \ =_{df}\ \mu X \bullet (P; X) \lhd b \rhd \text{II}$$

The fact that the underlying definition of the iteration construct is defined in terms of fix-points of recursive equations is highlighted by the fact it satisfies the unfolding rule.

Law 3.3.97. $(b * P) = (P; (b * P)) \lhd b \rhd \text{II}$

If the looping condition does not hold initially, the loop reduces to skip.

Law 3.3.98. $(\neg b)^{\top}_{S}; (b * P) = (\neg b)^{\top}_{S}$

Law 3.3.99. $(\neg b)^{\bot}_{S}; (b * P) = (\neg b)^{\bot}_{S}$

On the other hand, if the condition does hold, the loop's body gets executed at least once.

Law 3.3.100. $b^{\top}_{S}; (b * P) = b^{\top}_{S}; P; (b * P)$

On termination, the condition on which the iteration was looping is always **false**.

[3]Remember the synchronous theory requires that whole program to be **S**-healthy with respect to all variables in the system, regardless of whether they are in scope or not.

Law 3.3.101. $(b * P) = (b * P); (\neg b)^{\perp}_{S}$

If P preserves a given condition c, then so does its iterated form $(b * P)$.

Law 3.3.102. *Provided* $\left(c^{\perp}_{S}; P\right) = \left(c^{\perp}_{S}; P; c^{\perp}_{S}\right)$ *we have:*

$$c^{\perp}_{S}; (b * P) = c^{\perp}_{S}; (b * P); c^{\perp}_{S}$$

An iteration followed by a predicate Q can be expressed as a particular form of recursion.

Law 3.3.103. $(b * P); Q = \mu X \bullet (P; X) \lhd b \rhd Q$

The following result provides a very useful way of splitting the execution of a loop.

Law 3.3.104. $b * P; (b \vee q) * P = (b \vee q) * P$

It is possible to expand the scope of a variable from inside the body of a loop and this leads to refinement.

Law 3.3.105. *Provided P is* **S***-healthy and b does not mention x we have:*

$$(b * (\texttt{var}\ x; P; \texttt{end}\ x)) \sqsubseteq \texttt{var}\ x; (b * P); \texttt{end}\ x$$

3.3.8 Disjoint-alphabet parallelism

Up to this point, we have derived the effects of the programming operators in the synchronous theory by means of lifting their operands through healthiness conditions and then applying the basic definition of the operators from the design theory. In this way, we are able to derive theorems describing the result of the same operators in the synchronous theory. Unfortunately, the same approach cannot be followed in the case of the disjoint-alphabet parallel operator as there is no definition for it in the alphabetised predicate calculus. The solution is to provide a new definition for this operator in our theory.

Ideally, we would like to define the disjoint-alphabet parallel execution of programs $\mathbf{S}(P_1 \vdash P_2)$ and $\mathbf{S}(Q_1 \vdash Q_2)$ as $\mathbf{S}(P_1 \wedge Q_1 \vdash P_2 \wedge Q_2)$. Unfortunately, the presence of the healthiness conditions pre-empts the possibility of the alphabets of P and Q from being disjoint, as our definition would require. In fact, **S**-healthy designs necessarily share the variables c and $x.out$ for all x in the program's alphabet. On the other hand, the only context in which this operator will be used is within parallel by merge, after the shared variables have been renamed by means of the separating simulations. Thus, c and $x.out$ are shared among parallel processes; hence, they also need to be renamed. In this sense, a program

$$\mathbf{S}(P); U0(m, c, x.out)$$

can be reduced to the equivalent form

$$\mathbf{S}(P)[0.m, 0.c, 0.x.out/m', c', x.out']$$

As $\mathbf{S}(P) = P \wedge (x.in' = x.in) \wedge (c \leq c') \wedge (x.out \leq x.out')$, the renaming will not only modify the occurrences of m', c' and $x.out'$ in P but will also affect the healthiness conditions. To keep the presentation compact, we introduce the notation $P[\mathbf{i}]$ which denotes the renaming of all shared variables m' in predicate P to $i.m'$. More formally:

Definition 3.3.106. *Predicate renaming*

$$P[\mathbf{i}] =_{df} P[i.m'/m']$$

Based on the observations above and assuming that P and Q are renamed alphabet-disjoint designs, we can define the disjoint-alphabet parallel composition as follows:

Definition 3.3.107. *Disjoint-alphabet parallel execution of renamed* **S***-healthy designs*

$$\mathbf{S}[\mathbf{0}](P[\mathbf{0}]) \parallel \mathbf{S}[\mathbf{1}](Q[\mathbf{1}]) =_{df} \mathbf{S}[\mathbf{0}] \circ \mathbf{S}[\mathbf{1}](P[\mathbf{0}] \parallel Q[\mathbf{1}])$$

As with the selection operator, the application of the healthiness conditions can be separated from the application of the disjoint-parallel operator. In this way, the parallel operator is just combining designs (that will afterwards be made **S**-healthy). In this way, our new definition for disjoint parallel behaviour satisfies the same laws the parallel composition operator for designs does:

Law 3.3.108. $\mathbf{S}[\mathbf{0}](P[\mathbf{0}]) \parallel \mathbf{S}[\mathbf{1}](Q[\mathbf{1}]) = \mathbf{S}[\mathbf{1}](Q[\mathbf{1}]) \parallel \mathbf{S}[\mathbf{0}](P[\mathbf{0}])$

Law 3.3.109. $\mathbf{S}[\mathbf{0}](P[\mathbf{0}]) \parallel (\mathbf{S}[\mathbf{1}](Q[\mathbf{1}]) \parallel \mathbf{S}[\mathbf{2}](R[\mathbf{2}])) = (\mathbf{S}[\mathbf{0}](P[\mathbf{0}]) \parallel \mathbf{S}[\mathbf{1}](Q[\mathbf{1}])) \parallel \mathbf{S}[\mathbf{2}](R[\mathbf{2}])$

Law 3.3.110. $(\mathbf{S}[\mathbf{0}](P[\mathbf{0}]) \parallel \mathbf{S}[\mathbf{1}](I\!I_D[\mathbf{1}])) = \mathbf{S}[\mathbf{0}] \circ \mathbf{S}[\mathbf{1}](P[\mathbf{0}])$

Law 3.3.111. $\mathbf{S}[\mathbf{0}](P[\mathbf{0}]) \parallel \mathbf{S}[\mathbf{1}](\bot_D) = \mathbf{S}[\mathbf{0}] \circ \mathbf{S}[\mathbf{1}](\bot_D)$

Law 3.3.112. $\mathbf{S}(P \lhd b \rhd Q) \parallel \mathbf{S}[\mathbf{2}](R[\mathbf{2}]) = ((\mathbf{S}(P) \parallel \mathbf{S}[\mathbf{2}](R[\mathbf{2}])) \lhd b \rhd (\mathbf{S}(Q) \parallel \mathbf{S}[\mathbf{2}](R[\mathbf{2}])))$

Law 3.3.113. *For any descending chain* $S = \{S_n \mid n \in \mathbb{N}\}$ *we have that:*

$$(\bigsqcap S) \parallel \mathbf{S}[\mathbf{0}](R[\mathbf{0}]) = \bigsqcap_n (S_n \parallel \mathbf{S}[\mathbf{0}](R[\mathbf{0}]))$$

Law 3.3.114. *Provided x and y are different we have:*

$$\mathbf{S}[\mathbf{0}](x := e_1) \parallel \mathbf{S}[\mathbf{1}](y := e_2) = \mathbf{S}[\mathbf{0}] \circ \mathbf{S}[\mathbf{1}](x, y := e_1, e_2)$$

3.3.9 Parallel by merge

As already discussed in the previous section, the parallel execution of **S**-healthy designs needs to be able to handle that at least the c, $x.in$ and $x.out$ variables will be shared among them. In the case of parallel by merge, we have already described the mechanism of generating local copies of the shared variables for each process, produce their parallel execution by means of the alphabet-disjoint parallel operator and, afterwards, merge the local copies to calculate the final value of the

shared variables. Being able to treat the observations c and $x.out$ as shared variables, the definition for parallel-by-merge is mostly the same as it was in the theory of designs:

Definition 3.3.115. *Parallel by merge*

$$P \parallel_{\mathrm{M}} Q \;=_{df}\; ((P; U0(m, c, out)) \parallel (Q; U1(m, c, out)))_{+m,c,out} ; M$$

provided m and out are, respectively, the list of shared and all ".out" variables in the program

The parallel by merge operator in the synchronous theory satisfies a similar set of laws to its counterpart in the design theory. Provided P, Q and R are **S**-healthy, it is possible to show that $\parallel_{\mathrm{M}}$ is commutative, associative, it has II as a pseudo-unit and $\bot$ as its zero. It also distributes through the least-upper bound operator.

Law 3.3.116. $P \parallel_{\mathrm{M}} Q = Q \parallel_{\mathrm{M}} P$

Law 3.3.117. $P \parallel_{\mathrm{M}} (Q \parallel_{\mathrm{M}} R) = (P \parallel_{\mathrm{M}} Q) \parallel_{\mathrm{M}} R$

Law 3.3.118. $(\mathit{II} \parallel_{\mathrm{M}} P) \sqsubseteq P$

Law 3.3.119. $(\bot \parallel_{\mathrm{M}} P) = \bot$

Law 3.3.120. $(P \lhd b \rhd Q) \parallel_{\mathrm{M}} R = ((P \parallel_{\mathrm{M}} R) \lhd b \rhd (Q \parallel_{\mathrm{M}} R))$

Law 3.3.121. *For any descending chain $P = \{P_n \mid n \in \mathbb{N}\}$ we have that:*

$$(\bigsqcap P) \parallel_{\mathrm{M}} R = \bigsqcap (P \parallel_{\mathrm{M}} R)$$

It is possible to relate two assignments to the same variable being executed in asynchronous parallel by merge to an assignment using the selection function described in Section 2.3.8.1. In practical terms, this implies that we can turn parallel assignments to x into a single assignment of the form $x \overset{:=}{_{\mathrm{snc}}} e$ where e uses the selection function to merge the two updates. More formally:

Law 3.3.122. $x \overset{:=}{_{snc}} e_1 \parallel_{\mathrm{M}} x \overset{:=}{_{snc}} e_2 \sqsubseteq x \overset{:=}{_{snc}} \mathit{SELECT}(e_1, e_2, x)$

Finally, parallel by merge preserves **SH3**.

Theorem 3.3.123. *Parallel by merge – **SH3** preservation*

$$(P \parallel_{\mathrm{M}} Q); \mathit{II} = (P \parallel_{\mathrm{M}} Q)$$

provided P and Q are (**SH3**)*-healthy designs.*

3.3.10 Synchronous parallel by merge

To describe synchronous, parallel behaviour with shared variables we need to define how parallel processes synchronise with each other. The design theory already has a way to achieve this with the design **sync**. Our definition just lifts **sync** into the synchronous theory by making it **S**-healthy.

Definition 3.3.124. *The* **S***-healthy synchronisation predicate*

$$\mathsf{sync}(x) = \mathbf{S}(\boldsymbol{sync}(x))$$

As sync is just an assignment, it is easy to show that it is **SH3**-healthy.

Theorem 3.3.125. *The* sync *construct is* **SH3**

$$\mathbf{SH3}(\mathsf{sync})$$

On the other hand, extending the alphabet of the synchronisation action on a set of variables α ($\mathsf{sync}(\alpha)$) with the variable x is equivalent to a synchronisation over the extended alphabet $\alpha \cup \{x\}$.

Law 3.3.126. $\mathsf{sync}(\alpha)_{[x]} = \mathsf{sync}(\alpha \cup \{x\}) = \mathsf{sync}$

For the reminder of this section we will frequently need to refer to a shared variable x and its associated history variable $x.out$. In order to keep the presentation compact, we introduce the following notation.

Definition 3.3.127. *Shared variable with history*

$$\dddot{x} =_{df} \; x, x.out$$

Having a way for each process to signal when it is ready to synchronise and to keep a history of its updates to their copies of the shared variables, we now need to define the final merge predicate that will calculate how these individual updates get reflected in the actual shared variables.

This has already been addressed in Section 2.3.9 by means of the final merge predicate $\hat{M}$. Unfortunately, one of the requirements of the UTP definition of $\hat{M}$ has is that the two parallel processes must take the same amount of clock-cycles. As already pointed out in Section 3.1, we want our theory to allow the combination of any two processes in parallel, no matter how many clock cycles each of them take. In order to cope with this additional flexibility, we introduce a new definition of the final merge predicate.

Definition 3.3.128. *Final merge*

$$\begin{aligned}
&\hat{M}(\dddot{x}, \dddot{0.x}, \dddot{1.x}, \dddot{x'}) \; =_{df} \\
&\quad (0.x.out_{0.c}, 1.x.out_{1.c} := 0.x, 1.x); \\
&\quad \left(\begin{array}{l}
c := max(0.c, 1.c) \\
\wedge\, M(x_{0.c,1.c}, (0.x \lhd 0.c \geq 1.c \rhd x_{0.c,1.c}), (1.x \lhd 1.c \geq 0.c \rhd x_{0.c,1.c}), x') \\
\wedge\, M(x, 0.x.out_c, 1.x.out_c, x.out'_c) \lhd c < max(0.c, 1.c) \rhd (x.out'_c = x.out_c) \\
\wedge\, \{M(x.in_{i-1}, (0.out_i \lhd 0.c \geq i \rhd x.in_{i-1}), (1.out_i \lhd 1.c \geq i \rhd x.in_{i-1}), \\
\qquad x.out'_i \mid c < i < max(0.c, 1.c)\} \\
\wedge\, \{I\!I_{\{x.out_i\}} | i < c\})
\end{array} \right); \\
&\quad \boldsymbol{end}\; 0.c, 1.c, 0.x, 1.x, 0.x.out, 1.x.out
\end{aligned}$$

where $x_{0.c,1.c} = (x \lhd c = max(0.c, 1.c) \rhd x.in_{max(0.c,1.c)-1})$.

The above definition can be understood line-by-line as follows:

1. The first line in the definition above appends the final value of the local copy of x just after the last position in its 'history' variable. In this way, the merge predicate ensures that the last updates performed by the shorter process (if any) are not lost if the process terminates without synchronising. Although not necessary, this operation is performed over both processes and the reason for this redundancy is the fact that it is not possible, in general, to know which process will be the shortest of the two (i.e., take less clock cycles to terminate).

2. The second line in our definition of $\hat{M}$ deals with the merging of the clock cycle counter. Its merged value is set to the largest of the counters from the parallel processes.

3. In the third line, the merge predicate is used to calculate the value of the shared variable x. The reference value in the merge (i.e., the first argument to M) is the special value "$x_{0.c,1.c}$": it returns x if both processes terminate without performing any synchronisation, or the merged value for x in the clock cycle just before the parallel composition terminates otherwise.

   The rationale behind this choice is that the value of x is calculated by selecting the updated value (from either of the two parallel processes) that is different from the value held in x in the previous clock cycle. If both processes take zero clock cycles, the reference value for x is the value left in x by the process executing just before the parallel processes. If either of the processes takes at least one clock-cycle, the reference value for the merge predicate should be the merged value one clock cycle before the termination of the parallel processes.

   The second and third arguments to the merge predicate make sure the processes' local copies of x are not used by M, unless the process was actually active during the last clock cycle of the parallel execution. For example, consider the case of $P \parallel_{\hat{M}} Q$ where P takes four clock cycles to finish and Q takes only two. The final value of x should be calculated based on values in the local copies for x as produced at clock-cycle four. Given that Q finished two clock-cycles before that, it is not clear what the value of $1.x$ would be at clock-cycle four: its value is unconstrained. Our definition of $\hat{M}$ detects this problem (i.e., $1.c$ is clearly smaller than $max(0.c, 1.c)$ in this example) and solves it by selecting the value in $0.x$ to update x.

4. Lines four and five in our definition apply the merge predicate to all the intermediate synchronisation points that happened during the parallel execution. The values to be merged are taken from the local copies of $x.out_i$, as long as the corresponding process was actually executing during clock cycle i. Also, the first line in our definition appended the final value of x to the $x.out$ sequence for both processes. The way in which the arguments are passed to the merge predicate in line five makes M use this 'extended history' value if necessary. Using the same example again, when merging the values at clock cycle three, the value from Q will be $1.x$. This is precisely the value Q may have modified after its last synchronisation point at the end of clock cycle two. On the other hand, if Q did not perform any action after

synchronising at clock cycle two, then $1.x = x.in_2$ and $1.x.out_2 = x.in_2$. This is exactly the value that will make M select the value from P (i.e., the value stored in $0.x.out$).

5. Line six in our definition ensures that the values in the history previous to the execution of the parallel operator (i.e., those associated with clock-cycles lower than c) are kept constant.

6. Line seven finishes the scope of the local variables introduced by the separating simulations that are used as "working copies" by each of the parallel processes.

For some of the proofs in the following sections it will be useful to decompose our definition of $\hat{M}$ into the part that deals with updating the shared store x, the part that updates its observational history $x.out$ and the part that updates the clock cycle counter c. Our first definition below deals with the observational aspects of the final merge predicate updating $x.out$.

Definition 3.3.129.

$$
\begin{array}{l}
R_{hist}(x, 0.x, 1.x, x') =_{df} \\
\quad (0.x.out_{0.c}, 1.x.out_{1.c} := 0.x, 1.x); \\
\quad \left(\begin{array}{l}
\wedge\ M(x, 0.x.out_c, 1.x.out_c, x.out'_c) \lhd c < max(0.c, 1.c) \rhd (x.out'_c = x.out_c) \\
\wedge\ \{M(x.in_{i-1}, (0.out_i \lhd 0.c \geq i \rhd x.in_{i-1}), (1.out_i \lhd 1.c \geq i \rhd x.in_{i-1}), \\
\qquad x.out'_i \mid c < i < max(0.c, 1.c)\} \\
\wedge\ \{I\!I_{\{x.out_i\}} | i < c\})
\end{array} \right);
\end{array}
$$

The following definition allows us to separate all the aspects of $\hat{M}$ dealing with x and its associated history variable $x.out$.

Definition 3.3.130.

$$
\begin{array}{l}
R(x, 0.x, 1.x, x') =_{df} R_{hist}(x, 0.x, 1.x, x') \wedge \\
\qquad\qquad M(x_{0.c,1.c}, (0.x \lhd 0.c \geq 1.c \rhd x_{0.c,1.c}), (1.x \lhd 1.c \geq 0.c \rhd x_{0.c,1.c}), x')
\end{array}
$$

The following theorem shows the equivalence between our original formulation and the combination of the different aspects defined above.

Theorem 3.3.131. *Final merge alternative formulation*

$$\hat{M} = R(x, 0.x, 1.x, x'); c := max(0.c, 1.c); \textbf{\textit{end}}\ 0.c, 1.c, 0.x, 1.x, 0.x.out, 1.x.out$$

As with the design-based theory of synchronous processes, our final merge predicate satisfies all the properties in order to be a *valid* operator. First of all, $\hat{M}$ is symmetric on its input.

Law 3.3.132. $(\overset{...}{0.}x, \overset{...}{1.}x := \overset{...}{1.}x, \overset{...}{0.}x); \hat{M} = \hat{M}$

Provided we define

$$\hat{M}3 =_{df} \exists\ \ddot{x} \bullet \hat{M}(\ddot{x}, \overset{...}{0.}x, \overset{...}{1.}x, \ddot{x}) \wedge \hat{M}(\ddot{x}, \ddot{x}, \overset{...}{2.}x, \ddot{x'})$$

we have that $\hat{M}$ is associative in the sense of the following law.

Law 3.3.133. $(\overset{...}{0.}x, \overset{...}{1.}x, \overset{...}{2.}x{:=}\overset{...}{1.}x, \overset{...}{2.}x, \overset{...}{0.}x); \hat{M}3 = \hat{M}3$

We now can define the parallel execution of synchronous designs with shared variables. Provided m is the list of shared variables and P and Q are **S**-healthy, we define the synchronous parallel by merge operator $\|_{\hat{M}}$ as follows:

Definition 3.3.134. *Parallel by final merge*

$$P \|_{\hat{M}} Q =_{df} ((P; U0(c, \ddot{\vec{m}})) \parallel (Q; U1(c, \ddot{\vec{m}})))_{+c,\ddot{\vec{m}}}; \hat{M}$$

Our definition of the parallel-by-final-merge[4] operator allows us to prove a set of algebraic laws similar to the one we proved for the parallel combinator for disjoint processes. In particular, $\|_{\hat{M}}$ is commutative and associative, it has $\mathbb{I}$ as a pseudo-unit and $\bot$ as its zero. It also distributes through conditional and the least upper bound operator.

Law 3.3.135. $P \|_{\hat{M}} Q = Q \|_{\hat{M}} P$

Law 3.3.136. $P \|_{\hat{M}} (Q \|_{\hat{M}} R) = (P \|_{\hat{M}} Q) \|_{\hat{M}} R$

Law 3.3.137. $(\mathbb{I} \|_{\hat{M}} \mathbb{I}) = \mathbb{I}$

Law 3.3.138. $(\mathbb{I} \|_{\hat{M}} P) \sqsubseteq P$

Law 3.3.139. $(\bot \|_{\hat{M}} P) = \bot$

Law 3.3.140. $(P \lhd b \rhd Q) \|_{\hat{M}} R = ((P \|_{\hat{M}} R) \lhd b \rhd (Q \|_{\hat{M}} R))$

Law 3.3.141. *For any descending chain $S = \{S_n \mid n \in \mathbb{N}\}$ we have:*

$$(\bigsqcap S) \|_{\hat{M}} R = \bigsqcap_i (S_i \|_{\hat{M}} R)$$

The specialised distribution of sequential composition over $\|_{\hat{M}_D}$ described in Section 2.3.9 also holds in our theory.

Law 3.3.142. *Provided that neither P nor Q perform any* `sync` *actions we have:*

$$(P \|_M Q); \mathsf{sync}; (R \|_{\hat{M}} S) = (P; \mathsf{sync}; R) \|_{\hat{M}} (Q; \mathsf{sync}; S)$$

Assumption distributes rightwards over synchronous-parallel by merge.

Law 3.3.143. $b_S^\top; (P \|_{\hat{M}} Q) = (b_S^\top; P) \|_{\hat{M}} (b_S^\top; Q)$

If one of the parallel **S**-healthy designs finishes by performing an assertion of the form $(x = e)_S^\bot$ and the other branch has its alphabet extended to incorporate x, then the whole program can be simplified by extracting the assertion and performing it after the parallel region terminates.

[4]Notice that the definition of parallel-final-merge is nothing but parallel-by-merge with a larger set of shared variables (we now need to account for c and $x.out$) and a different merge predicate (i.e., one that can handle the merging of the additional variables).

Law 3.3.144. $(P; (x = e)^{\perp}_{S}) \parallel_{\hat{M}} Q_{[x]} = (P \parallel_{\hat{M}} Q_{[x]}); (x = e)_{\perp}$

An assignment that synchronises with the environment can be composed in parallel-by-final-merge with sync without affecting the semantics of the program.

Law 3.3.145. $(x \mathrel{:=}_{snc} e; \mathit{sync}) = (x \mathrel{:=}_{snc} e; \mathit{sync}) \parallel_{\hat{M}} \mathit{sync}$

The following law allows the simplification of the synchronous parallel-by-final-merge execution of a pair assignments when certain conditions are met.

Law 3.3.146. *Provided x and y are different variables and that e_2 does not depend on x we have:*

$$(x \mathrel{:=}_{snc} e_1; \mathit{sync}) \parallel_{\hat{M}} (y \mathrel{:=}_{snc} e_2; \mathit{sync}) = (x, y \mathrel{:=}_{snc} e_1, e_2); \mathit{sync}$$

For simplicity reasons, we have kept track of the shared alphabet implicitly and used the symbol $\parallel_{\hat{M}}$ to denote the parallel-by-final-merge operator over a shared alphabet m. In the remaining part of this section we will use the notation $\parallel^{m}_{\hat{M}}$ to refer to the parallel-by-final-merge with shared alphabet m. In these terms, extending the alphabet of $(P \parallel_{\hat{M}} Q)$ to include a fresh variable x is refined by extending the alphabet of P and Q and treating x as a shared variable.

Law 3.3.147. *Provided P and Q are* **S***-healthy we have:*

$$(P \parallel^{m}_{\hat{M}} Q)_{[x]} \sqsubseteq (P_{[x]} \parallel^{m,x}_{\hat{M}} Q_{[x]})$$

Extending the scope of a variable that is local to one of the parallel branches to encompass the whole parallel program leads to refinement.

Law 3.3.148. *Provided P and Q are* **SH3***-healthy, we have:*

$$(\mathit{var}\ x; P; \mathit{end}\ x) \parallel^{m,c}_{\hat{M}} Q \sqsubseteq \mathit{var}\ x; (P \parallel^{m,c,x}_{\hat{M}} Q_{[x]}); \mathit{end}\ x$$

3.3.11 Guarded commands

In general, the notation $(b \rightarrow P)$ stands for Nelson's notion of guarded commands [Nelson 1989]: if the guard b is true, the whole command behaves like P; otherwise, it behaves miraculously. Our notion of a synchronous guarded command $(b \overset{\rightarrow}{_{snc}} P)$ construct is defined in a slightly different way: it behaves like Nelson's in the case where the condition holds; it does nothing but synchronise with the environment otherwise.

Definition 3.3.149. *Synchronous guarded command*

$$b \overset{\rightarrow}{_{snc}} P \ =_{df}\ P \lhd b \rhd \mathit{sync}$$

provided P is **S***-healthy.*

The following two laws characterise algebraically the result of executing a guarded command in environments where the value of its condition is known.

Law 3.3.150. $(b_S^\top ; b \mathrel{\overrightarrow{}}_{snc} P) = (b_S^\top ; P)$

Law 3.3.151. $((\neg b)_S^\top ; b \mathrel{\overrightarrow{}}_{snc} P) = ((\neg b)_S^\top ; \mathtt{sync})$

The nested application of guarded commands can be simplified to a single guarded command where the condition is the conjunction of the two guards.

Law 3.3.152. $b_1 \mathrel{\overrightarrow{}}_{snc} b_2 \mathrel{\overrightarrow{}}_{snc} P = (b_1 \wedge b_2) \mathrel{\overrightarrow{}}_{snc} P$

The parallel-by-merge combination of guarded commands is itself a guarded command.

Law 3.3.153. *Provided P and Q are of the form $(v \mathrel{:=}_{snc} e; \mathtt{sync})$ we have:*

$$b \to P \parallel_{\hat{M}} c \to Q = (b \vee c) \mathrel{\overrightarrow{}}_{snc} (b \to P \parallel_{\hat{M}} c \to Q)$$

The parallel by merge combination of several guards can be simplified provided certain conditions are met.

Law 3.3.154. *Provided $\neg(b \wedge c)$ and P is of the form $(v \mathrel{:=}_{snc} e; \mathtt{sync})$ we have:*

$$b_S^\top ; ((b \mathrel{\overrightarrow{}}_{snc} P) \parallel_{\hat{M}} (c \mathrel{\overrightarrow{}}_{snc} Q)) = b_S^\top ; b \mathrel{\overrightarrow{}}_{snc} P$$

Law 3.3.155. *Provided P and Q are of the form $(v \mathrel{:=}_{snc} e; \mathtt{sync})$ we have:*

$$b \mathrel{\overrightarrow{}}_{snc} (P \parallel_{\hat{M}} Q) = (b \mathrel{\overrightarrow{}}_{snc} P) \parallel_{\hat{M}} (b \mathrel{\overrightarrow{}}_{snc} Q)$$

Law 3.3.156. *Provided P is of the form $(v \mathrel{:=}_{snc} e; \mathtt{sync})$ we have:*

$$(b_1 \mathrel{\overrightarrow{}}_{snc} P) \parallel_{\hat{M}} (b_2 \mathrel{\overrightarrow{}}_{snc} P) = (b_1 \vee b_2) \mathrel{\overrightarrow{}}_{snc} P$$

Law 3.3.157. *Provided P and Q are of the form $(v \mathrel{:=}_{snc} e; \mathtt{sync})$ we have:*

$$((s \wedge b) \mathrel{\overrightarrow{}}_{snc} P) \parallel_{\hat{M}} ((s \wedge \neg b) \mathrel{\overrightarrow{}}_{snc} Q) = s \mathrel{\overrightarrow{}}_{snc} (P \lhd b \rhd Q)$$

Provided c holds and that P preserves c, the condition of a loop of the form $b * P$ can be extended to $(b \wedge c)$.

Law 3.3.158. *Provided $(c_S^\top ; P) = (c_S^\top ; P ; c_S^\top)$ and that P is* **S** *healthy, we have:*

$$c_S^\top ; b * P = c_S^\top ; (b \wedge c) * P$$

Extending the alphabet of $(b \mathrel{\overrightarrow{}}_{snc} P)$ is equivalent to extending the alphabet of P.

Law 3.3.159. $(b \mathrel{\overrightarrow{}}_{snc} P)_{[x]} = b \mathrel{\overrightarrow{}}_{snc} P_{[x]}$

3.3.12 Feedback loop

One of the main reasons for introducing the theory of synchronous designs was to be able to assert/assume the value assigned to a variable across synchronisation points when there are no other parallel processes modifying the variable. More formally, we want the theory of synchronous designs to satisfy the following law stating that, when no other process is modifying x, we have that:

$$(x \overset{:=}{_{\mathsf{snc}}} e; \mathsf{sync}) = (x \overset{:=}{_{\mathsf{snc}}} e; \mathsf{sync}; (x = e)^{\top}_{S})$$

In Section 3.1 we claimed the above result can be proved in the presence of the feedback loop. So far, we have defined the feedback loop between $x.in$ and $x.out$ as the equation $x.in = x.out'$. The intuition behind this definition is that the equality linking $x.in$ to $x.out'$ is a point-wise equality over all indexes in the sequence. For the results in this section, we will sometimes need to address this point-wise equality in a more precise way:

Definition 3.3.160. *Sequence equality*

$$s \overset{=}{_{[i\,..\,k]}} t \;=_{df} \bigwedge_{j \in [i\,..\,k]} s_j = t_j$$

With sequence equality defined in this way it is possible to formulate a number of properties that will be useful in this and later sections. Provided $i \leq k$, a sequence equality expression in the range $[0\,..\,k]$ always encompasses an expression over the smaller range $[0\,..\,i]$.

Law 3.3.161. *Provided $i \leq k$ we have:*

$$(s \overset{=}{_{[0\,..\,i]}} t) \wedge (s \overset{=}{_{[0\,..\,k]}} t) = (s \overset{=}{_{[0\,..\,k]}} t)$$

Provided that $0 \leq i \leq k$, a sequence equality expression over the range $[0\,..\,k]$ can be partitioned into two sequence-equality expressions over the ranges $[0\,..\,i]$ and $[i\,..\,k]$.

Law 3.3.162. *Provided $i \leq c_0$ and $c_0 \leq k$ we have:*

$$(s \overset{=}{_{[i\,..\,k]}} t) = (s \overset{=}{_{[i\,..\,c_0]}} t) \wedge (s \overset{=}{_{[c_0\,..\,k]}} t)$$

Extending the range in a sequence equality expression places additional restrictions over the program and, hence, it leads to refinement.

Law 3.3.163. *Provided $i \leq j \leq k$ we have:*

$$(s \overset{=}{_{[i\,..\,j]}} t) \sqsubseteq (s \overset{=}{_{[i\,..\,k]}} t)$$

Returning to our goal of proving that the results of an assignment can be assumed after the synchronisation action, which now can be formalised and proved.

Law 3.3.164. $(x \overset{:=}{_{\mathsf{snc}}} e; \mathsf{sync}) \wedge (x.in \overset{=}{_{[0\,..\,c']}} x.out') = (x \overset{:=}{_{\mathsf{snc}}} e; \mathsf{sync}; (x = e)^{\top}_{S}) \wedge (x.in \overset{=}{_{[0\,..\,c']}} x.out')$

As mentioned in the beginning of this chapter, all programs in the theory of synchronous designs have an implicit feedback loop of the form $(x.in_{[0\,..\,\overset{=}{c'}]} x.out')$ establishing the kind of link we need between $x.in$ and $x.out$. Nevertheless, the feedback loop defined in this way is only enough to conduct the proof above provided the program is comprised of a single assignment. If we are to handle more general programs, we need a way to distribute the effects of the feedback loop throughout the program. Distributing the feedback loop over conditional is straightforward, as shown by the following law.

Law 3.3.165. $(P \lhd b \rhd Q) \wedge (x.in_{[0\,..\,\overset{=}{c'}]} x.out') = (P \wedge (x.in_{[0\,..\,\overset{=}{c'}]} x.out')) \lhd b \rhd (Q \wedge (x.in_{[0\,..\,\overset{=}{c'}]} x.out'))$

The case of sequential composition is, however, more complicated. The main problem of the feedback sequential composition lies in the restriction that the feedback loop must hold, even if the synchronous design fails to be started or its precondition is not satisfied. In this situation, the fact that the healthiness conditions in the synchronous theory are also required to hold in the non-successful cases is what allows us to prove the following law:

Law 3.3.166. *Provided P and Q are* **S***-healthy designs we have:*

$$(P; Q) \wedge (x.in_{[0\,..\,\overset{=}{c'}]} x.out') = P \wedge (x.in_{[0\,..\,\overset{=}{c'}]} x.out'); Q \wedge (x.in_{[0\,..\,\overset{=}{c'}]} x.out')$$

It is sometimes useful to be able to extend the scope of a feedback loop from an **S**-healthy design Q to include another design P when they are in sequential composition $P; Q$. The rationale behind this law is that the feedback loop is binding $x.in$ to $x.out'$ over all clock-cycles from the beginning of the whole program in which P and Q are included (i.e., clock-cycle count 0). This is the key to prove the following law.

Law 3.3.167. *Provided P and Q are* **S***-healthy designs we have:*

$$P; (Q \wedge (x.in_{[0\,..\,\overset{=}{c'}]} x.out')) = P \wedge (x.in_{[0\,..\,\overset{=}{c'}]} x.out'); Q \wedge (x.in_{[0\,..\,\overset{=}{c'}]} x.out')$$

The following law allows us to take advantage of the fact that P is a **SH3**-healthy design in order to eliminate $\text{I\!I}$ whilst still preserving the feedback loop.

Law 3.3.168. *Provided P is a* **SH3***-healthy design, we have:*

$$P; \text{I\!I} \wedge (x.in_{[0\,..\,\overset{=}{c'}]} x.out') = P \wedge (x.in_{[0\,..\,\overset{=}{c'}]} x.out')$$

The feedback loop also distributes through iteration.

Law 3.3.169. *Provided P is* **S** *and* ***H3*** *we have:*

$$((b * P) \wedge (x.in_{[0\,..\,\overset{=}{c'}]} x.out')) = b * (P \wedge (x.in_{[0\,..\,\overset{=}{c'}]} x.out'))$$

With the set of laws above it is possible to distribute the feedback loop over all programming constructs in our language except the parallel combinators. That is because the definition of parallel-by-merge relies on the $x.in$ sequence being bound to its actual values after the parallel

execution has been resolved and the values for *x.out* have already been merged. The consequence of not having such a law, allows us to distribute the feedback loop over non-parallel fragments of the program only. This result is reassuring as we have already pointed out the laws that need the feedback loop can only take place outside parallel regions of the program. Based on the previous results, it is also possible to show that provided $(x = e)$ holds before the execution of $P_{[x]}$ then it is possible to assume the same condition after $P_{[x]}$.

Law 3.3.170. *Provided P is* **S***-healthy, P does not mention x and x is not an observational variable we have:*

$$((x = e)^{\top}_{S}; P_{[x]}) \wedge (x.in_{[0 \,..\, c']}^{=} x.out') = ((x = e)^{\top}_{S}; P_{[x]}; (x = e)^{\top}_{S}) \wedge (x.in_{[0 \,..\, c']}^{=} x.out')$$

Similarly, provided it is possible to assert $(x = e)$ after the execution of the loop $(b * P)_{[x]}$ as shown by the following law.

Law 3.3.171. *Provided P is* **SH3** *healthy we have:*

$$\begin{aligned}((x = e)^{\top}_{S}; (b * P)_{[x]}) \wedge (x.in_{[0 \,..\, c']}^{=} x.out') \sqsubseteq \\ ((x = e)^{\top}_{S}; (b \wedge x = e) * P_{[x]}) \wedge (x.in_{[0 \,..\, c']}^{=} x.out')\end{aligned}$$

3.4 Chapter summary

This chapter covered the following topics:

- **Limitations of the design theory as the semantic framework for synchronous, parallel programs with shared-variables**. The main issues with the design theory as the semantic framework for the reasoning language are related with its limited capacity for assertional reasoning when trespassing clock cycle boundaries.

- **New theory for reasoning about synchronicity**. New observations for keeping track of the clock-cycle and history of each of the variables in the program where introduced. The new theory is completed by means of healthiness conditions ensuring that: (a) time only progresses forward; (b) the variable's history is preserved; and (c) the synchronisation mechanism is only controlled by the theory rather than by any of the processes.

- **Programming and reasoning constructs for synchronous programs**. Following the UTP approach, all constructs in the new theory were obtained by means of making the corresponding designs satisfy the healthiness conditions of the synchronous theory.

- **Basic operators in the reasoning language**. Most operators in the new theory (i.e., sequential composition, selection, iteration) use the same definitions provided for them in alphabetised relational calculus. This chapter shows these operators are closed under the healthiness conditions of the synchronous theory and provided theorems that described how to calculate their results.

- **Synchronous parallel-by-merge**. A new definition for parallel-by-merge (together with a new merge operator) was introduced in order to allow the theory to cope with parallel processes that take different amounts of clock-cycles to terminate.

- **Synchronous alphabet extension**. The need to account for the updates in the history of the variables in the extended alphabet led to the introduction of the new alphabet extension. The new definition states that $P_{[x]}$ not only finishes by requesting that x remains constant, but also that x's history reflects this same fact during all clock cycles in which P has been executing.

- **Synchronous guarded commands**. Our definition is similar to Nelson's guarded commands but synchronises with the environment in case the condition does not hold. This last fact not only captures the synchronous nature of our domain in a better way, it also allows the usage of parallel-by-merge as the combinator for guarded commands in our compilation normal forms (see Section 5.1).

- **Extended set of reasoning laws**. On top of re-validating all laws presented for the theory of designs, new results were proved in the synchronous theory accounting for: assertional reasoning over clock boundaries; alphabet extension for shared-variables, parallel construct; dynamic scope in synchronous contexts; guarded command combination and simplification.

The constructs, operators and algebraic laws presented in this chapter serve as the core reasoning language for our compiler. The next chapter applies this theory to provide semantics for Handel-C and introduces additional constructs needed for reasoning about loops and communications.

Chapter 4

Handel-C and its reasoning language

"*If you are faced by a difficulty or a controversy in science, an ounce of algebra is worth a ton of verbal argument.*"

— J.B.S. Haldane

The synchronous theory introduced in the previous chapter provides the definitions of the operators and constructs needed for basic reasoning about synchronous programs. In this chapter we focus on showing how the synchronous theory forms the core of our reasoning language and provide extensions for reasoning about special constructs in the context of our compiler for Handel-C.

One of the main aspects of this chapter is the introduction of denotational semantics for Handel-C. The formalism we have adopted for this task is the theory of synchronous designs described in the previous section. The semantics is, in turn, used in three main ways: (a) to discover and prove properties about Handel-C programs; (b) to prove equivalences between different constructs in the language that will be useful in the compilation chapters; and (c) to establish the link between Handel-C and the reasoning language.

The other objective of this chapter is to introduce additional reasoning constructs that will be used in the compilation process. On top of the extended constructs we have already presented in the previous section (e.g., abort, miracle, assertions and assumptions) this section introduces additional operations that allow a finer degree of control over iteration, communications and prioritised choice. The main importance of these extended constructs is that they allow the splitting of the complex behaviour of the communications and priorities primitives into simpler actions. Furthermore, the formulation in terms of the new constructs allows the operational details of input/output and **priAlt** to be abstracted out from the algebraic formulation. In this way, we can provide compilation laws for these constructs that are elegant and simpler to verify.

4.1 Handel-C semantics in the UTP

Handel-C programs are comprised of at least one **main** function and, possibly, some additional functions. Multiple main functions within the same program produce the parallel execution of their bodies under the same clock domain.

All C-based constructs in Handel-C behave as defined in ANSI-C [Kernighan and Ritchie 1988] but with some additional restrictions regarding the clock-based, synchronous nature of the language. In this sense, the evaluation of expressions is performed by means of combinatorial circuitry and it is completed within the clock cycle in which it is initiated (expressions are considered to be evaluated "for free" [Celoxica Ltd. 2002*a*] due to this semantic interpretation). Assignment, on the other hand, happens at the end of the clock cycle.

The fact that the assignment construct modifies the variables at the end of the clock cycle makes the updated value available to be read by other processes by the beginning of the next clock cycle. This notion is equivalent to the synchronisation action in the theory of synchronous designs (the definition of `sync` not only makes the results available to other processes, but also signals the end of the current clock cycle). Based on this observation, the semantics of assignment in Handel-C are described by the following definition.

Definition 4.1.1. *Handel-C assignment*

$$[\![x \mathrel{:=_{HC}} e]\!] =_{df} x \mathrel{:=_{snc}} [\![e]\!] ; \texttt{sync}$$

The sequential composition of two Handel-C programs P and Q takes no time to transfer the control to Q once P has finished. This behaviour can be precisely described by sequential composition in the UTP, as captured by the following definition.

Definition 4.1.2. *Sequential composition in Handel-C*

$$[\![P \fatsemi Q]\!] =_{df} [\![P]\!] ; [\![Q]\!]$$

The combinatorial way of evaluating conditions affects the timing of all the constructs in Handel-C. In the case of selection, the branch selected for execution (depending on the condition) will start execution within the same clock cycle in which the whole construct is initiated. In the UTP we can achieve the same semantic effect by means of the conditional construct, as shown by the following definition.

Definition 4.1.3. *Conditional in Handel-C*

$$[\![\textbf{\textit{if}}\ c\ \textbf{\textit{then}}\ P\ \textbf{\textit{else}}\ Q]\!] =_{df} [\![P]\!] \lhd [\![c]\!] \rhd [\![Q]\!]$$

While active, the **while** construct starts executing its body in the same clock cycle its condition evaluates to true. Similarly, it terminates within the same clock cycle in which its condition becomes false. As with the sequential composition and conditional constructs, the semantics can be defined in terms of the equivalent UTP construct.

Definition 4.1.4. *While construct in Handel-C*

$$[\![\textbf{\textit{while}}\ b\ \textbf{\textit{do}}\ P]\!] =_{df} [\![b]\!] * [\![P]\!]$$

Handel-C also provides other, non C-based constructs meant to take advantage of the underlying synchronous hardware the program is targeting. The most basic construct in this category is the **delay** construct, that leaves the state unchanged but takes a whole clock cycle to terminate. This behaviour is easily stated in our theory by a process that behaves like Π and then synchronises with the environment. Due to the fact that skip is a left unit for sequential composition, we can simplify the semantic expression for **delay** and define it as follows.

Definition 4.1.5. *Handel-C delay*

$$[\![\textbf{\textit{delay}}]\!] =_{df} \texttt{sync}$$

The remaining, non-C constructs in Handel-C behave like they do in CSP [Hoare 1983]. In the case of parallel composition, statements are executed in a *true* parallel way as they denote independent pieces of hardware running within the same clock domain. As expected, the semantics of this construct are given in terms of the synchronous-parallel operator defined in the previous chapter.

Definition 4.1.6. *Parallel composition in Handel-C*

$$[\![P \parallel_{HC} Q]\!] =_{df} [\![P]\!] \parallel_{\hat{M}} [\![Q]\!]$$

Finally, input and output have the standard blocking semantics: if the two parts are ready to communicate, the value outputted at one end is assigned to the variable associated with the input side. Both sides of the communication take one full clock cycle to successfully communicate. A process trying to communicate over a channel without the other side being ready will block (delay) for a single clock cycle and try again.

In the semantics, we define the input and output commands to rely upon a set of special variables that are not included in the list of program variables. The special set of variables associated to a given channel ch include $ch?, ch!$ and ch standing, respectively, for the requests for inputting, outputting and the value to be transmitted over ch. We also assume that $ch?, ch!$ (the requests for communication) will remain in the logical value **false** unless they are used. This assumption is consistent with the hardware implementation of communications, where the requests are wires that remain in a "low state" unless they are explicitly fed with current when the request is done.

We also introduce the fixed, but arbitrary value `ARB`. As with the **false** logical value for the communication requests, this value will be the default value for all channels when they are not being used. This is the same kind of refinement implemented at the hardware level where the value of this kind of bus is supposedly unconstrained but physically implemented by means of having all wires comprising the bus being set to *low*. One important remark is that `ARB` is a value outside the type of values being transmitted over the bus. This observation will be used in the reduction to second normal form described in Chapter 6.

The final merge predicate presented in Chapter 3 accounts for these modifications by taking the initial value at the beginning of the clock cycle as the reference value when merging wires. This is exactly the same principle used for the "standard" variables in the program (we compare

against $x.in_{i-1}$ which is the value a given to each variable x at the beginning of each clock cycle i). In this case, the value assigned at the beginning of the clock cycle is constant and known, so we can simplify the equations to include this knowledge.

In this context, the semantics of the input/output primitives can be stated as follows:

Definition 4.1.7. *The input construct*

$$[\![ch?m]\!] =_{df} \mu X \bullet ch? \overset{:=}{_{snc}} \mathbf{true}; ((m \overset{:=}{_{snc}} ch.in_c; \mathtt{sync}) \lhd ch!.in_c = \mathbf{true} \rhd \mathtt{sync}; X)$$

Definition 4.1.8. *The output command*

$$[\![ch!e]\!] =_{df} ch!, ch \overset{:=}{_{snc}} \mathbf{true}, [\![e]\!]; \mu X \bullet (\mathtt{sync} \lhd ch?.in_c = \mathbf{true} \rhd \mathtt{sync}; ch!, ch \overset{:=}{_{snc}} \mathbf{true}, [\![e]\!]; X)$$

The semantics of the prioritised choice construct is defined by a two-level semantic function. At the top level, we introduce the variable **res** as a mean to capture the current state of the **priAlt** construct. The **res** variable holds the value **true** if a previous guard was ready to communicate (and hence, the **priAlt** is ready to terminate); otherwise, it holds the value **false**. At the hardware level, **res** is implemented as a chain of communication requests, where the next request is placed only if the previous request was not granted.

Definition 4.1.9. *Top level semantics for the* ***priAlt*** *construct*

$$[\![\mathbf{priAlt}\,\{P\}]\!] =_{df}$$
$$\mathbf{var}\ \mathbf{res} \overset{:=}{_{snc}} \mathbf{false}; \mu X \bullet [\![P]\!]; (\mathit{II} \lhd \mathbf{res} \rhd \mathtt{sync}; X); \mathbf{end}\ \mathbf{res}$$

Even though each of the case expressions within a **priAlt** is syntactically like input/output command, they do not behave like those constructs. *Input/output guards* (i.e., when input/output commands are used as conditions in case expressions) behave like input/output commands if the communication is possible. Otherwise, the whole case expression reduces to skip and the control is transferred to the next case statement.

The lower level semantics deal with each of the case statements inside a **priAlt**. Naturally, the semantic expressions depend on the external variable **res** for their execution. If the external variable indicates the **priAlt** is ready to terminate, the whole case statement behaves like skip. Otherwise, it verifies the case's condition and updates **res** according to the result of the evaluation.

In the case of input/output guards, the actions performed in the semantics are similar to the ones presented when describing input/output commands outside the **priAlt** construct but without the recursive call.

Definition 4.1.10. *Semantics for input guards*

$$[\![\mathbf{case}\ ch?m\mathbf{:}\ P \fatsemi \mathbf{break}]\!] =_{df}$$
$$\mathit{II} \lhd \mathbf{res} \rhd (ch? \overset{:=}{_{snc}} \mathbf{true}; (\mathbf{res}, m \overset{:=}{_{snc}} \mathbf{true}, ch.in_c; \mathbf{sync}; [\![P]\!]_{+\mathbf{res}} \lhd ch!.in_c \rhd \mathit{II}))$$

Definition 4.1.11. *Semantics for output guards*

$$[\![\textbf{\textit{case}}\ ch!val\textbf{\textit{:}}\ P \fatsemi \textbf{\textit{break}}]\!] =_{df}$$
$$\Pi \lhd \mathbf{res} \rhd (ch!, ch \mathrel{:=}_{snc} \textbf{\textit{true}}, val; (\mathbf{res} \mathrel{:=}_{snc} \textbf{\textit{true}}; \textbf{\textit{sync}}; [\![P]\!]_{+\mathbf{res}} \lhd ch?.in_c \rhd \Pi))$$

Finally, executing a default guard when the **priAlt** is still to be resolved produces the termination of the **priAlt** (i.e., it sets **res** to **true** and executes the actions associated to it).

Definition 4.1.12. *Semantics for the default clause*

$$[\![\textbf{\textit{default:}}\ P]\!] =_{df} \Pi \lhd \mathbf{res} \rhd (\mathbf{res} \mathrel{:=}_{snc} \textbf{\textit{true}}; [\![P]\!]_{+\mathbf{res}})$$

As mentioned before, the shallow embedding we have used to give semantics to Handel-C allows us to take direct advantage of most of the laws we have already shown in the synchronous designs theory. Being able to prove laws that are consistent with the operational intuition behind Handel-C operators provides reassuring evidence that our semantic model correctly captures the actual behaviour of the constructs in the language.

The rest of this section is devoted to showing some of the laws that can be proved from the semantics as defined above. Sequential composition is associative, it has Π as left unit and $\bot$ as left zero.

Law 4.1.13. $P \fatsemi (Q \fatsemi S) = (P \fatsemi Q) \fatsemi S$

Law 4.1.14. $(\Pi \fatsemi P) = P$

Law 4.1.15. $(\bot \fatsemi P) = \bot$

Parallel composition is commutative, associative and it has Π as 'pseudo-unit' (i.e., it leads to refinement) and $\bot$ as zero.

Law 4.1.16. $P \parallel_{HC} Q = Q \parallel_{HC} P$

Law 4.1.17. $P \parallel_{HC} (Q \parallel_{HC} S) = (P \parallel_{HC} Q) \parallel_{HC} S$

Law 4.1.18. $(P \parallel_{HC} \Pi) \sqsubseteq P$

Law 4.1.19. $(P \parallel_{HC} \bot) = \bot$

It is possible to extract an assignment out of a parallel branch, provided the other branches delay or perform other assignments.

Law 4.1.20. $x \mathrel{:=}_{HC} e \fatsemi (P \parallel_{HC} Q) = (x \mathrel{:=}_{HC} e \fatsemi P) \parallel_{HC} (\textbf{\textit{delay}} \fatsemi Q)$

Law 4.1.21. $x, y \mathrel{:=}_{HC} e_1, e_2 \fatsemi (P \parallel_{HC} Q) = (x \mathrel{:=}_{HC} e_1 \fatsemi P) \parallel_{HC} (y \mathrel{:=}_{HC} e_2 \fatsemi Q)$

When communication is possible, it can be replaced by assignment.

Law 4.1.22. $(ch?x \fatsemi P) \parallel_{HC} (ch!e \fatsemi Q) = (x, ch?, ch!, ch \mathrel{:=}_{HC} e, true, true, e) \fatsemi (P \parallel_{HC} Q)$

Law 4.1.23. $(ch?x \fatsemi P) \parallel_{HC} (ch!e \fatsemi Q) \parallel_{HC} (ch?y \fatsemi R) = (x, y, ch?, ch!, ch \coloneqq_{HC} e, e, true, true, e) \fatsemi (P \parallel_{HC} Q \parallel_{HC} R)$

An arbitrary-depth **priAlt** with a default clause can be expressed in terms of simpler binary **priAlt**s with default guards.

Law 4.1.24.
$$\left[\!\left[\textbf{\textit{priAlt}} \left\{ \begin{array}{l} \textbf{\textit{case}}\ ch_1?x\textbf{:}\ P_1 \fatsemi \textbf{\textit{break}} \\ \langle guard\_list \rangle \fatsemi \\ \textbf{\textit{default:}}\ P_n \end{array} \right. \right]\!\right] = \left[\!\left[\textbf{\textit{priAlt}} \left\{ \begin{array}{l} \textbf{\textit{case}}\ ch_1?x\textbf{:}\ P_1 \fatsemi \textbf{\textit{break}} \\ \textbf{\textit{default:}}\ \left[\!\left[\textbf{\textit{priAlt}} \left\{ \begin{array}{l} \langle guard\_list \rangle \fatsemi \\ \textbf{\textit{default:}}\ P_n \end{array} \right. \right]\!\right] \end{array} \right. \right]\!\right]$$

A **priAlt** construct with a single input (output) guard reduces to a single input (output) command.

Law 4.1.25. *Provided that P is a* **SH3**, **S***-healthy design and it does not mention* **res** *we have that:*

$$\textbf{\textit{priAlt}}\,\{\textbf{\textit{case}}\ ch?x\textbf{:}\ P \fatsemi \textbf{\textit{break}}\} = ch?x \fatsemi P$$

4.2 The reasoning language

The synchronous theory introduced in Chapter 3 provides the definitions and semantic domain for our reasoning language. Furthermore, the laws and theorems proved for the synchronous theory constitute the basic axioms from where we can show the compilation theorems of Handel-C programs into hardware descriptions.

As the constructs and laws in the reasoning language have already been presented in the thesis (Chapters 1 and 3 respectively), the focus of this section is on defining and providing semantics to the additional constructs that are used in the reduction of Handel-C programs to normal form. We begin by introducing a compact notation that allows us to explicitly control the advance in the clock cycle counter. In section 4.2.2, we introduce an extended iteration operator that allows us not only to specify what the body of the loop is, but also to specify which actions are to be performed after the looping condition becomes **false**. In the next chapter we require this kind of *iterating selection* to perform at least one time-consuming action once the looping condition becomes **false**. This more restrictive form of the complete iteration is the only kind of loop our compiler accepts in its input.

The next topic of this section deals with the input and output commands. The behaviour of these constructs is very complex, as it involves recursion and the recursion depends on whether the communication was successful or not. To address this issue, we introduce additional constructs to the language that, together, account for the behaviour of the input/output commands. In this way, it is possible to consider each of the independent aspects of an input/output command in isolation by means of decomposing them into more primitive actions. The key advantage of this approach is that it allows us to separate the algebraic aspects of the input/output constructs from their semantic expressions.

Finally, we introduce a particular form of the case statement. Our modified version of the case construct allows us to perform some actions before checking for the condition for each individual case. As the conditions for each case statement are based on the possibility of performing communications over channels, we can take advantage of the primitive actions we already introduced for

input/output commands. The key contribution of this subsection is the possibility of expressing the **priAlt** construct in terms of the primitives for input/output together with our modified case construct.

4.2.1 Timed constructs

It is common practice when reasoning about timed systems to have a construct that explicitly represents the action of advancing the clock cycle. In the synchronous theory, this can be easily achieved by means of performing a `sync` action as described in the following definition.

Definition 4.2.1. *One clock cycle skip*

$$\mathit{II}_1 =_{df} \texttt{sync}$$

From the definition above, it is straightforward to show that it is equivalent to the semantic expression associated to the **delay** construct, as shown by the following theorem.

Theorem 4.2.2. $\mathit{delay} = \mathit{II}_1$

Similarly, we can define an assignment that takes one clock cycle to finish as shown by the following definition.

Definition 4.2.3. *One clock cycle assignment*

$$(x \overset{:=}{_{snc}} e)_1 =_{df} (x \overset{:=}{_{snc}} e); \texttt{sync}$$

As in the case of the **delay** construct, the one-clock-cycle assignment defined above is equivalent to assignment in Handel-C.

Theorem 4.2.4. $x \overset{:=}{_{HC}} e = (x \overset{:=}{_{snc}} e)_1$

From the above definitions it is possible to show that the law allowing us to assume (assert) the value assigned to a variable is also valid for the timed assignment defined above provided the variable is local.

Law 4.2.5. $(\texttt{var}\ x; P; (x := e)_1; Q; \texttt{end}\ x) = (\texttt{var}\ x; P; (x := e)_1; (x = e)^{\top}_S; Q; \texttt{end}\ x)$

Law 4.2.6. $(\texttt{var}\ x; P; (x := e)_1; Q; \texttt{end}\ x) = (\texttt{var}\ x; P; (x := e)_1; (x = e)^{\bot}_S; Q; \texttt{end}\ x)$

These results are similar to law 3.3.164 but we have removed the requirement of having an explicit feedback loop for x as we are in the synchronous designs theory and all variables have an implicit feedback loop associated to them. The variable's local scope ensures there is no other process updating variable x, eliminating the possible conflicts arising from parallel processes updating the same variable.

4.2.2 Iterating selection

When programming in the imperative paradigm, programs rarely perform iterations as their final construct. Furthermore, it is very common to find that some actions are performed after the execution of loops in order to collect or manipulate the results produced while the iteration was executing.

In order to capture this frequent combination of constructs we introduce a new operator in our reasoning language. The new operator is an extended form of the selection construct that iterates its *then* branch while the selecting condition holds. As soon as the condition becomes false, the iterating selection transfers control to its *else* branch and terminates.

Definition 4.2.7. *Iterating selection*

$$b * (P) \blacktriangleright Q \;=_{df}\; (b * P); Q$$

Notice that by means of law 3.3.103, the above definition can be expressed in terms of the least fix point operator as follows:

Theorem 4.2.8.

$$b * (P) \blacktriangleright Q \;=_{df}\; \mu X \bullet (P; X) \lhd b \rhd Q$$

The fact that the iterating selection construct behaves like a loop while its condition is **true** and switches to the other branch as soon as the looping condition becomes **false** is fully characterised by the following algebraic laws:

Law 4.2.9. $(b)^{\top}; b * (P) \blacktriangleright Q = (b)^{\top}; P; b * (P) \blacktriangleright Q$

Law 4.2.10. $(\neg b)^{\top}; b * (P) \blacktriangleright Q = (\neg b)^{\top}; Q$

4.2.3 Communication primitives

Input and output in Handel-C have a complex behaviour that implements blocking semantics by means of attempting the communication at the beginning of their execution. If the communication is possible, the value provided as input gets transmitted over internal buses and assigned to the variable waiting for it at the other end of the channel.

As expected, the transfer of values only takes place provided the processes at both ends of the channel are ready to communicate. The arbitration mechanism in charge of detecting that both processes are ready to communicate is defined in terms of dedicated wires that signal a process' readiness to perform an input/output operation. In section 4.1 we took advantage of this operational intuition behind the communication primitives to define the semantics of the input and output constructs.

Our first intention when defining the reasoning language for these constructs was to use an approach based on the wires used to signal and control the input and output mechanisms. Unfortunately, the application of this idea to individual input/output commands in the program leads to a

non-functioning program. To see why, consider the case where we apply the replacement to one of the input commands in the program. Immediately after this, all other output commands in the program will stop being able to interact with it as they operate at a different level of abstraction (i.e., they do not refer to wires, they operate at the *channel level*). Furthermore, these transformations will have to be coordinated with the introduction of data refinement, as declarations of channels will have to be replaced by the underlying means used for the communication (i.e., wires used in the arbitration mechanism and buses for the transfer of information).

One possible solution to this problem is to perform all the above mentioned transformations (data and control refinement) at once to the program as a whole (in a way similar to the one proposed in Morgan's refinement calculus [Morgan 1990]). Even though this approach produces the effect we want while preserving the functionality of the program, it has two disadvantages: (a) it is not compositional (as we need to address program as a whole); and (b) it is not possible to verify its correctness regarding our semantics for Handel-C (as we are replacing the constructs by the semantic expressions themselves!).

The fundamental problem with the approaches mentioned above is that we are being forced to perform too many transformations in a single step (i.e., data and control refinement) when we only intend to achieve a higher degree of operational control over the actions performed by input/output commands (i.e., we only intend to perform control refinement). The reason for this coupled refinement lies in the fact that our more detailed operational control over the communication primitives are formulated in terms of the wires and buses used in the implementation.

The solution is to formulate primitives that still operate over channels (rather than over the underlying wires and buses) but that address the different actions that comprise the whole behaviour of the input and output constructs. From this observation, the first operation we want to introduce represents the action of one side of the channel (either the reader or the writer) being ready to communicate. We capture these actions with the new commands: **in-req**(ch) and **out-req**(ch). Following the notation introduced in section 4.1, we define the semantics of these new constructs as follows:

Definition 4.2.11. *Input request semantics*

$$[\![\textbf{\textit{in-req}}(ch)]\!] =_{df} ch? \overset{:=}{_{snc}} \textbf{\textit{true}}$$

Definition 4.2.12. *Output request semantics*

$$[\![\textbf{\textit{out-req}}(ch)]\!] =_{df} ch! \overset{:=}{_{snc}} \textbf{\textit{true}}$$

On the other hand, we also introduce a way of checking whether the reader (writer) is present at the other end of a channel ch. We denote these new constructs with the boolean conditions **rd**(ch) and **wr**(ch) respectively. The following definitions provide a more precise semantics for these two constructs:

Definition 4.2.13. *Input-ready condition*

$$[\![\boldsymbol{rd}(ch)]\!] =_{df} ch?.in_c = \boldsymbol{true}$$

Definition 4.2.14. *Output-ready condition*

$$[\![\boldsymbol{wr}(ch)]\!] =_{df} ch!.in_c = \boldsymbol{true}$$

Finally, we introduce a command that represents the two actions involved in the communication over a channel ch. The operation **in**(ch) returns the value being transmitted over channel ch at the present clock cycle (i.e., once the circuit has reached a stable state) and the command **out**(ch, e) denotes the action of sending the value e over channel ch.

Definition 4.2.15. *Primitive input*

$$[\![\boldsymbol{in}(ch)]\!] =_{df} ch.in_c$$

Definition 4.2.16. *Primitive output*

$$[\![\boldsymbol{out}(ch, e)]\!] =_{df} ch \overset{:=}{_{snc}} [\![e]\!]$$

It is important to notice the differences between the standard Handel-C input construct $ch\mathbf{?}x$ and the new **in**(ch) primitive. The former is a complex command that is able to handle the cases where the communication is possible (effectively assigning the input value to x) as well as the case when there is no input over ch during the current clock cycle (by performing a one-clock cycle delay and attempting the communication again). The latter, on the other hand, captures the value over the channel regardless of whether there has been an input to the channel or not (retrieving an undefined value). Moreover, **in**(ch) only models the action of reading the current value held in the channel ch. There is no notion of assigning this value to a variable as in $ch\mathbf{?}x$.

A similar difference holds between $ch\mathbf{!}e$ and **out**(ch, e). In the case of **out**(ch, e), the value e is forced into the channel ch without any verification of whether there is another process to receive the value or not (i.e., it implements a non-blocking, unbuffered channel). On the other hand, $ch\mathbf{!}e$ follows a protocol that ensures blocking semantics of communication over channel ch.

These differences are precisely characterised by the following equivalence laws relating Handel-C's input and output commands to particular combinations of the primitives described above.

Law 4.2.17. $ch\mathbf{?}x = \mu X \bullet \boldsymbol{in\text{-}req}(ch); ((x := \boldsymbol{in}(ch))_1 \lhd \boldsymbol{wr}(ch) \rhd \boldsymbol{delay}; X)$

Law 4.2.18. $ch\mathbf{!}e = \mu X \bullet \boldsymbol{out\text{-}req}(ch); \boldsymbol{out}(ch, e); (\boldsymbol{delay} \lhd \boldsymbol{rd}(ch) \rhd \boldsymbol{delay}; X)$

The basic communication primitives described above satisfy a number of additional algebraic properties. Firstly, it is possible to commute the order between a request and its opposite condition:

Law 4.2.19. $(\boldsymbol{wr}(ch))^{\top}_{S}; \boldsymbol{in\text{-}req}(ch) = \boldsymbol{in\text{-}req}(ch); (\boldsymbol{wr}(ch))^{\top}_{S}$

Law 4.2.20. $(\boldsymbol{rd}(ch))^{\top}_{S}; \boldsymbol{out\text{-}req}(ch) = \boldsymbol{out\text{-}req}(ch); (\boldsymbol{rd}(ch))^{\top}_{S}$

The **out**(ch, e) command does not affect the reading portion of the channel:

Law 4.2.21. $(\neg \textbf{\textit{rd}}(ch))^{\top}_{S}; \textbf{\textit{out}}(ch, e) = (\neg \textbf{\textit{rd}}(ch))^{\top}_{S}; \textbf{\textit{out}}(ch, e); (\neg \textbf{\textit{rd}}(ch))^{\top}_{S}$

Provided that a request or output operation is the first action on both branches of a conditional, it can be pulled out of the conditional construct (note that the condition in the selection construct does not depend on the requests being extracted from the branches).

Law 4.2.22. $(\textbf{\textit{in-req}}(ch); P) \lhd \textbf{\textit{wr}}(ch) \rhd (\textbf{\textit{in-req}}(ch); Q) = \textbf{\textit{in-req}}(ch); (P \lhd \textbf{\textit{wr}}(ch) \rhd Q)$

Law 4.2.23. $(\textbf{\textit{out-req}}(ch); P) \lhd \textbf{\textit{rd}}(ch) \rhd (\textbf{\textit{out-req}}(ch); Q) = \textbf{\textit{out-req}}(ch); (P \lhd \textbf{\textit{rd}}(ch) \rhd Q)$

Law 4.2.24. $(\textbf{\textit{out}}(ch, e); P) \lhd \textbf{\textit{rd}}(ch) \rhd (\textbf{\textit{out}}(ch, e); Q) = \textbf{\textit{out}}(ch, e); (P \lhd \textbf{\textit{rd}}(ch) \rhd Q)$

To keep the presentation compact, we will take advantage of the symmetry of input/output and for the remaining of this section **DIR** will represent a generic direction that remains constant for the whole extent of the law in which it appears.

Sequences of more than one request over the same channel may arise during the same clock cycle. Their joint effect is the same as the one corresponding to the last one of them.

Law 4.2.25. $\neg \textbf{\textit{DIR-req}}(ch); \textbf{\textit{DIR-req}}(ch) = \textbf{\textit{DIR-req}}(ch)$

Law 4.2.26. $\textbf{\textit{DIR-req}}(ch); \neg \textbf{\textit{DIR-req}}(ch) = \neg \textbf{\textit{DIR-req}}(ch)$

A request for input/output over a channel *ch* followed by a one-clock-cycle assignment that does not depend on *ch* can be performed in parallel.

Law 4.2.27. $\textbf{\textit{in-req}}(ch); (v \underset{snc}{:=} e)_1 = \textbf{\textit{in-req}}(ch)_1 \parallel_{\hat{M}} (v \underset{snc}{:=} e)_1$

Law 4.2.28. $\textbf{\textit{out-req}}(ch); (v \underset{snc}{:=} e)_1 = \textbf{\textit{out-req}}(ch)_1 \parallel_{\hat{M}} (v \underset{snc}{:=} e)_1$

The following laws allow us to eliminate the request for input (output) and its associated condition.

Law 4.2.29. *Provided there are no external requests for communication over ch we have that:*

$$(s_1 \vee s_2) \underset{snc}{\rightarrow} \textbf{\textit{in-req}}(ch)_1 \parallel_{\hat{M}} b \underset{snc}{\rightarrow} (v \underset{snc}{:=} e)_1 = b \underset{snc}{\rightarrow} (v \underset{snc}{:=} e)_1[(s_1 \vee s_2)/\textbf{\textit{rd}}(ch)]$$

Law 4.2.30. *Provided there are no external requests for communication over ch we have that:*

$$(s_1 \vee s_2) \underset{snc}{\rightarrow} \textbf{\textit{out-req}}(ch)_1 \parallel_{\hat{M}} b \underset{snc}{\rightarrow} (v \underset{snc}{:=} e)_1 = b \underset{snc}{\rightarrow} (v \underset{snc}{:=} e)_1[(s_1 \vee s_2)/\textbf{\textit{wr}}(ch)]$$

Law 4.2.31. *Provided that ch = ARB, and there are no external requests for communication over ch we have that:*

$$s_1 \underset{snc}{\rightarrow} \textbf{\textit{out}}(ch, e_1)_1 \parallel_{\hat{M}} s_2 \underset{snc}{\rightarrow} \textbf{\textit{out}}(ch, e_2)_1 \parallel_{\hat{M}} b \underset{snc}{\rightarrow} (v \underset{snc}{:=} e)_1 \sqsubseteq$$
$$b \underset{snc}{\rightarrow} (v \underset{snc}{:=} e)_1[\mathit{SELECT}(\mathit{ARB}, (e_1 \lhd s_1 \rhd \mathit{ARB}), (e_2 \lhd s_2 \rhd \mathit{ARB}))/\textbf{\textit{in}}(ch)]$$

At the hardware level, wires that are not latched acquire the **false** logical value at the end of the clock cycle. They remain in this state unless a process explicitly sets them to the **true** logical state. The following two laws capture this notion for the case where no process is setting the wires to the **true** value:

Law 4.2.32. *Provided there are no external* ***in-req***(ch) *events we have that:*

$$(v \overset{:=}{_{snc}} e)_1 = (v \overset{:=}{_{snc}} e)_1[\mathit{false}/\mathbf{rd}(ch)]$$

Law 4.2.33. *Provided there are no external* ***out-req***(ch) *events we have that:*

$$(v \overset{:=}{_{snc}} e)_1 = (v \overset{:=}{_{snc}} e)_1[\mathit{false}/\mathbf{wr}(ch)]$$

4.2.4 Nested conditionals: the case statements

In this section we are interested in expressing multi-alternative selection constructs that contain a list of guarded commands that are checked in a waterfall way. In this sense, it easy to see that a sequence of case statements of the form

$$\textbf{case } b_1\textbf{: } P_1 \fatsemi \textbf{break} \fatsemi \textbf{case } b_2\textbf{: } P_2 \fatsemi \textbf{break}$$

can be seen as a nested conditional expression

$$P_1 \lhd b_1 \rhd (P_2 \lhd b_2 \rhd \mathit{II})$$

Even though the above observation would allow us to describe the behaviour of constructs like `switch` in C and `ALT` in Occam [Barrett 1992] we are interested in defining a more general construct where each case statement contains a sequence of actions to be executed before evaluating the case's condition. Once the actions are executed, the case's condition is evaluated and the associated program is executed provided the condition holds. If the guard is not true, control is transferred and the second process gets activated. More formally, we define a case statement as follows:

Definition 4.2.34. *Case statement*

$$\mathit{case}\ a; cond\ ?\ P \mid Q \ =_{df}\ a; (P \lhd cond \rhd Q)$$

It is also possible to have compound cases that are triggered in a 'waterfall' way:

Definition 4.2.35. *Nested case*

$$\mathit{case}\ a_1; c_1\ ?\ P_1 \mid (\textbf{case}\ a_2; c_2\ ?\ P_2 \mid Q) \ =_{df}\ a_1; (P_1 \lhd c_1 \rhd a_2; (P_2 \lhd c_2 \rhd Q))$$

Based on the possibility of writing nested cases, we introduce the generalised case statement, that allows an arbitrary number of guarded actions:

Definition 4.2.36. *Generalised case*

$$\textbf{\textit{case}} \begin{pmatrix} a_1; g_1 \ ?\ P_1 \mid \\ \vdots \\ a_n; g_n \ ?\ P_n \mid \\ P \end{pmatrix} =_{df} \textbf{\textit{case}}\ a_1; g_1 \ ?\ P_1 \mid (\textbf{case}\ a_2; g_2 \ ?\ P_2 \mid ... \mid \textbf{case}\ a_n; g_n \ ?\ P_n \mid P)$$

From the definitions above, we can prove a number of algebraic laws that fully characterise the behaviour of the case statement. The case where the first guard is known to hold, resolves immediately, performing the prelude-actions followed by the actions in the true part of the condition:

Law 4.2.37. *Provided* $(g_1)^\top_S$ *commutes with a we have that:*

$$g^\top; \textbf{\textit{case}}\ (a; g \ ?\ P_1 \mid P_2) = g^\top; a; P_1$$

Similarly, we can simplify the case of a guarded statement when the first condition is known to not to hold. In this case, only the prelude-actions of the first case statement are executed, followed by the expression in the **false** branch of the conditional.

Law 4.2.38. *Provided* $(\neg g_1)^\top_S$ *commutes with a we have that:*

$$(\neg g)^\top; \textbf{\textit{case}}\ (a; g \ ?\ P_1 \mid P_2) = (\neg g)^\top; a; P_2$$

The following two results are particular cases of laws 4.2.37 and 4.2.38 where the actions in the false branch are themselves case expressions.

Law 4.2.39. *Provided* $(g_1)^\top_S$ *commutes with* a_1 *we have that:*

$$(g_1)^\top_S; \textbf{\textit{case}}\ (a_1; g_1 \ ?\ P_1 \mid \cdots \mid a_n; g_n \ ?\ P_n \mid P) = (g_1)^\top_S; a_1; P_1$$

Law 4.2.40. *Provided* $(\neg g_1)^\top_S$ *commutes with* a_1 *we have that:*

$$(\neg g_1)^\top_S; \textbf{\textit{case}}\ (a_1; g_1 \ ?\ P_1 \mid a_2; g_2 \ ?\ P_2 \mid \ldots) = (\neg g_1)^\top_S; a_1; \textbf{\textit{case}}\ (a_2; g_2 \ ?\ P_2 \mid \ldots)$$

4.2.4.1 Prioritised choice

The **priAlt** operator in Handel-C behaves like a C `switch` construct but with two fundamental differences: (a) the conditions activating each case statement are based on the possibility of communicating values over some of the program's channels; and (b) if none of the cases was allowed to execute and there was no default guard, the whole construct delays for a whole clock cycle and tries again.

Even though *input/output guards* in case expressions differ in behaviour from their counterparts outside the **priAlt** construct, there is still the need for the arbitration mechanism to determine whether the communication can take place or not. In this sense, both input/output commands and guards still need to signal to the environment their readiness to communicate over their channels

and also need to be able to receive the outcome of the arbitration mechanism to either activate their associated programs or to transfer control to the subsequent case statement. As expected, the atomic actions used to decompose input and output commands in the previous section allow us to describe the actions within case statements.

Following the approach introduced in the previous section, we will abstract from the *direction* in which each channel is used by each individual case in order to keep the presentation compact and to be able to concentrate in the algebraic treatment of the **priAlt** construct. In this sense we introduce the function *req* that captures the requests corresponding to each of the possible communication guards inside a **priAlt** construct. More precisely, we define the function *req* as follows:

Definition 4.2.41.

$$req(ch?x) =_{df} \textbf{\textit{in-req}}(ch)$$
$$req(ch!e) =_{df} \textbf{\textit{out-req}}(ch); \textbf{\textit{out}}(ch, e)$$

We also need a function to extract the condition that makes an input/output command hold when used as a guard inside a **priAlt** construct. We capture this notion in the function *chk*:

Definition 4.2.42.

$$chk(ch?x) =_{df} \textbf{\textit{wr}}(ch)$$
$$chk(ch!e) =_{df} \textbf{\textit{rd}}(ch)$$

As the above definitions are just wrappers over the primitive actions introduced for input/output commands, they satisfy a similar set of properties to their counterparts in the previous section. We only present here the ones that will be needed in later sections to treat *req* and *chk* at the algebraic level, without the need for expanding their definitions.

The function *req* and an assumption based on *chk* commute when put in sequence (note that as they refer to the same communication g, this law is just summarising laws 4.2.19 and 4.2.20).

Law 4.2.43. $(chk(g))^{\top}; req(g) = req(g); (chk(g))^{\top}$

If we are requesting the communication associated with a communication guard g regardless of the value of the communicating condition, then the request can be extracted from the conditional:

Law 4.2.44. $(req(g); P) \lhd chk(g) \rhd (req(g); Q) = req(g); (P \lhd chk(g) \rhd Q)$

Finally, we need a function mapping an input/output guard into the actions that are performed when the guard holds **true**. The function *act* captures this notion:

Definition 4.2.45.

$$act(ch?x) =_{df} (x \overset{:=}{_{snc}} \textbf{\textit{in}}(ch))_1$$
$$act(ch!e) =_{df} \mathbb{I}_1$$

Armed with these definitions, we can state the main equivalence laws between the **priAlt** construct and the modified case expression introduced in this section.

Law 4.2.46. *Prialt with default clause equivalence*

$$\mathbf{priAlt}\left\{\begin{array}{l}\mathbf{case}\ g_1\mathbf{:}\ P_1\ ;\ \mathbf{break};\\ \vdots\\ \mathbf{case}\ g_n\mathbf{:}\ P_n\ ;\ \mathbf{break};\\ \mathbf{default:}\ P_d\end{array}\right\} = \mathbf{case}\left(\begin{array}{l}req(g_1); chk(g_1)\ ?\ act(g_1); P_1\ |\\ \vdots\\ req(g_n); chk(g_n)\ ?\ act(g_n); P_n\ |\\ P_d\end{array}\right)$$

Law 4.2.47. *Prialt without default clause equivalence*

$$\mathbf{priAlt}\left\{\begin{array}{l}\mathbf{case}\ g_1\mathbf{:}\ P_1\ ;\ \mathbf{break};\\ \vdots\\ \mathbf{case}\ g_n\mathbf{:}\ P_n\ ;\ \mathbf{break}\end{array}\right\} = \mu X \bullet \mathbf{case}\left(\begin{array}{l}req(g_1); chk(g_1)\ ?\ act(g_1); P_1\ |\\ \vdots\\ req(g_n); chk(g_n)\ ?\ act(g_n); P_n\ |\\ I\!I_1; X\end{array}\right)$$

4.3 Chapter summary

In this chapter we have covered the following topics:

- **Semantics for Handel-C**. Semantics for all constructs in the input language were given in the context of the theory of synchronous designs described in Chapter 3. Several properties about Handel-C programs and equivalence laws between different constructs were shown to hold from the semantics. The fact that the semantics are given in the same framework used to give semantics to the reasoning language builds the link between the Handel-C and its representation within the reasoning language.

- **The reasoning language**. The reasoning language presented in Chapter 1 was given semantics within the theory of synchronous designs. Furthermore, the basic reasoning laws are those proved in Chapter 3. An extended set of reasoning construct was also presented in this chapter with the goal of (a) simplify the notation in the reasoning language; and (b) provide a mechanism to gain more control over communications and prioritised choice without including implementation details in the reasoning language. Finally, equivalence laws were provided among the new primitives and the original Handel-C communication constructs.

In the next chapter we address the compilation of Handel-C within the framework of the reasoning language. The proofs and definitions in the reminder of the thesis are conducted purely by algebraic means, as all semantic details have been enclosed within this and the previous chapters.

Part III

Reducing Handel-C to netlists

Chapter 5

Normal forms and compilation

"*The primary goal of theorists [...] is to prove a collection of laws which is sufficiently comprehensive that any other inequation between programs can be derived from the laws alone by algebraic reasoning*"

— C.A.R. Hoare and Jifeng He

The aim of this work is to produce a compiler from Handel-C into its corresponding, and semantically equivalent, normal form implementation. In the context of our compilation process, each normal form is a state machine representation of the original Handel-C program. Our first normal form captures the source program control state by means of a set of control variables. In turn, each of the constructs in the program is represented as a parallel set of actions guarded by the appropriate combination of the control variables.

In our second (and final) normal form we split the calculation of the values of expressions from the process of updating the program variables with those values. Wires are introduced to capture and transfer the value of expressions (intended to be calculated by means of combinatorial logic at the hardware level) into the memory-capable devices used to implement the program's variables.

5.1 First normal form: one clock cycle parallel choice

The goal of our first normal form is to capture Handel-C's control flow in a state machine. We achieve this effect by means of introducing new variables $a_1, \ldots, a_n$ and associating the program's control states to different values for them.

Each construct in the program is then represented as a set of one-clock cycle long *steps*. Let a be the list of the machine control variables, P be a one-clock cycle long predicate describing the effect of the step both, over a and the store of the machine; and k be the state of the normal form described by this step, then we can formally define a *step* as follows:

Definition 5.1.1. *Step*

$$(a = k) \underset{snc}{\rightarrow} P$$

In practical terms, our formulation of steps as defined above can only represent very simple Handel-C programs (i.e., those comprised by a single assignment or a **delay** construct). In order to capture more complex behaviour, we need a way of representing combinations of steps. This can be achieved by means of providing a way of representing the choice between two or more steps. In general, the choice between two guarded commands is denoted by

$$b \underset{\text{snc}}{\rightarrow} P_1 \ \Box \ c \underset{\text{snc}}{\rightarrow} P_2$$

and it provides a very elegant way of modelling a given mechanism that can deterministically perform *only one* of n possible actions, depending on its current state. The necessary condition, however, is that all guards must be disjoint. In case the guards are not disjoint, there is the chance that more than one guard will become true at a given time, leading to the introduction of non-determinism in the system.

This way of combining steps has been successfully used in the compilation of sequential programming languages [Nelson and Manasse 1992; Sampaio 1993; Silva et al. 1997*b*; Duran et al. 2001]. From this observation it is easy to see that this choice operator will also be sufficient to describe the C-based aspects of Handel-C by means of associating a unique value to each control state and guarding each step with the appropriate control-based conditions. In this way, a sequence of assignments of the form

$$x \underset{\text{HC}}{:=} e_1 \ ; \ y \underset{\text{HC}}{:=} e_2$$

can be represented as the state machine

$$\begin{array}{l} \texttt{var}\ a; a \underset{\text{snc}}{:=} s_1; \\ \quad (a = s_1 \vee a = s_2) * (a = s_1 \underset{\text{snc}}{\rightarrow} (x, a \underset{\text{snc}}{:=} e_1, s_2)_1) \ \ \Box \ \ (a = s_2 \underset{\text{snc}}{\rightarrow} (y, a \underset{\text{snc}}{:=} e_2, f)_1)\,; \\ \texttt{end}\ a \end{array}$$

Handel-C also allows the description of parallel programs. In the context of our formulation in terms of state machines, this means several steps activating at the same time. Modelling this kind of behaviour is clearly not possible in the context of the choice operator described above given its inherently sequential nature. One possible alternative is to capture the parallel execution of steps

$$b \underset{\text{snc}}{\rightarrow} P \parallel_{\hat{M}} c \underset{\text{snc}}{\rightarrow} Q$$

by means of combining the actions P and Q in parallel and guarding this new parallel step with the conjunction of b and c as follows:

$$(b \wedge c) \underset{\text{snc}}{\rightarrow} (P \parallel_{\hat{M}} Q)$$

This solution has been adopted for the compilation of Occam into Hardware [He et al. 1993] but, unfortunately, it has two main disadvantages:

1. Even though the actions within every step take advantage of the real parallelism available in the hardware, the state machine capturing the Handel-C program being compiled is still sequential in nature (i.e., only one step is active at any given clock cycle).

2. The solution described above is encoding parallelism by producing a product state machine of $(b \overrightarrow{}_{\text{snc}} P)$ and $(c \overrightarrow{}_{\text{snc}} Q)$. Even though this is a mathematically elegant solution, it produces an exponential increment in the number of control states and, consequently, in the number of steps in the resulting machine. This exponential increase is the consequence of the explicit representation of all possible combinations between the steps of the two parallel machines that this technique needs to produce. This solution is not acceptable if we take into account that the usual target for Handel-C programs are FPGA devices with limited physical resources available.

Our solution is to promote the parallel behaviour to the state machine level and allow the possibility of several steps to be active at any given time. A natural way of implementing this idea is by means of replacing the choice operator mentioned above with the parallel by merge construct:

Definition 5.1.2. *Parallel step combination*

$$(b \overrightarrow{}_{snc} P) \parallel_{\hat{M}} (c \overrightarrow{}_{snc} Q)$$

Due to the way in which we have defined our guarded commands (if the condition does not hold, the whole guarded command reduces to `sync`), if b and c above are disjoint, our combination operator behaves exactly like the traditional choice operator described earlier in this section. In this way we are not only able to cater for parallel behaviour (by means of having more than one step active at a given time) but also to describe sequential behaviour when the conditions in the steps are disjoint.

Finally, thanks to Law 3.3.153 we know that steps are closed under the $\parallel_{\hat{M}}$ combinator (i.e., the $\parallel_{\hat{M}}$-combination of steps is itself a step). This property allows us to postulate our lemmas and theorems for a single step and refer to binary (or more complex) steps only when needed for clarity purposes.

5.1.1 Reasoning about steps

As expected, the *activation* of a given step can only occur if the machine is in a control state satisfying the step's guard. When the step is a compound step (i.e., a product of the combination of two or more steps), its activation is the set of states that may trigger the activation of any of the individual steps that comprise it. More formally, we define the activations for a step $b(a_1, a_2, \ldots, a_n) \overrightarrow{}_{\text{snc}} A$ as the set of values for the control variables $a_1, a_2, \ldots, a_n$ that would make condition b hold.

Definition 5.1.3. *Activations*

$$act\,(b(a_1, a_2, \ldots, a_n) \overrightarrow{}_{snc} A) \;=_{df}\; \{\langle v_1, v_2, \ldots, v_n\rangle \mid b(v_1, v_2, \ldots, v_n)\}$$

Similarly, the set of *continuations* of a step are those values that the control variables can take after the execution of the step. If $A(a'_1, \ldots, a'_n, z')$ is the action part of the step, $a'_1, \ldots, a'_n$ are the final values of the machine's control variables after the execution of A and z is the list of program variables; the set of continuations are the possible after values for the control variables $a'_1, \ldots, a'_n$ that could be produced by the execution of A.

Definition 5.1.4. *Continuations*

$$cont\left(b \underset{snc}{\rightarrow} A(a'_1, a'_2, \ldots, a'_n, z')\right) =_{df} \{\langle v_1, \ldots, v_n\rangle \mid A(v_1, v_2, \ldots, v_n, z')\}$$

As mentioned before, guarding steps with disjoint conditions is the key to achieving non-parallel behaviour. Two steps P and Q that do not share any activation state cannot be active at the same time and are regarded as *disjoint*. More formally, we define two steps to be disjoint as follows:

Definition 5.1.5. *Disjointness*

$$disj(P, Q) =_{df} act(P) \cap act(Q) = \{\}$$

When a step P pre-empts the execution of another step Q we say that P *inhibits* Q. This notion can be easily formalised by requiring that the continuations of P are disjoint from the activations of Q, as shown by the following definition.

Definition 5.1.6. *Inhibition*

$$inh(P, Q) =_{df} cont(P) \cap act(Q) = \{\}$$

Algebraically, in a context where a is the list of control variables, a step P inhibiting another step Q can be characterised by the following fix-point equivalence:

Law 5.1.7. *Provided $inh(P, Q)$, then we have:*

$$inh(P, Q) \Rightarrow P = P; (a \notin act(Q))^{\perp}_{S}$$

5.1.2 Iterated steps

So far we have presented ways to combine several steps together and to analyse the possible interaction among them. In this section we concentrate on defining the concept of iteration for a set of steps and analysing its properties.

The iteration of a set of steps connected by the parallel choice operator (Definition 5.1.2) is defined by a loop that has the set of steps as its body and the disjunction of their guards as its looping condition.

Definition 5.1.8. *Step iteration*

$$*((b \underset{snc}{\rightarrow} P) \parallel_{\hat{M}} (c \underset{snc}{\rightarrow} Q)) = b \vee c * ((b \underset{snc}{\rightarrow} P) \parallel_{\hat{M}} (c \underset{snc}{\rightarrow} Q))$$

Being itself a loop, the iteration of steps satisfies the unfolding mechanism.

Law 5.1.9. $*(b \rightarrow_{snc} P) = (P \lhd b \rhd \mathit{II}); *(b \rightarrow_{snc} P)$

Due to Law 3.3.153 the result above also applies to sets of parallel steps (as the whole set of steps can be represented as a single step). Furthermore, if the conditions of the set of steps are disjoint, we can further simplify the Law 5.1.9 as described by the following law.

Law 5.1.10. *Provided* $\neg(b \wedge c)$ *then we have:*

$$b_S^\top; *((b \rightarrow_{snc} P) \parallel_{\hat{M}} (c \rightarrow_{snc} Q)) = b_S^\top; P; *((b \rightarrow_{snc} P) \parallel_{\hat{M}} (c \rightarrow_{snc} Q))$$

The body of an iterated step with condition b can be extended with another step $c \rightarrow_{snc} Q$ provided b and c cannot hold simultaneously.

Law 5.1.11. *Provided* $\neg(b \wedge c)$ *and* P *takes at least one clock cycle, then we have:*

$$*(b \rightarrow_{snc} P) = b * ((b \rightarrow_{snc} P) \parallel_{\hat{M}} (c \rightarrow_{snc} Q))$$

If the body of an iterated step is self-inhibiting, the loop can be altogether eliminated.

Law 5.1.12. *Provided* $inh(b \rightarrow_{snc} P, b \rightarrow_{snc} P)$ *we have:*

$$*(b \rightarrow_{snc} P) = P \lhd b \rhd \mathit{II}$$

The following law shows how information about activation and inhibition of steps can be extracted outside of iterated steps.

Law 5.1.13. *Provided* $inh(P, Q)$ *then we have:*

$$(a \notin act(Q))_S^\perp; *P = (a \notin act(Q))_S^\perp; *P; (a \notin (act(Q) \cup act(P))_S^\perp$$

The joint iteration of steps $b \rightarrow_{snc} P$ and $c \rightarrow_{snc} Q$ can be serialised (by means of performing the iteration of the former step first) provided $c \rightarrow_{snc} Q$ inhibits $b \rightarrow_{snc} P$.

Law 5.1.14. *Provided* $\neg(b \wedge c)$ *and* $inh(c \rightarrow_{snc} Q, b \rightarrow_{snc} P)$ *then we have:*

$$*((b \rightarrow_{snc} P) \parallel_{\hat{M}} (c \rightarrow_{snc} Q)) = *(b \rightarrow_{snc} P); *(c \rightarrow_{snc} Q)$$

The result above can be further extended provided we know that $b \rightarrow_{snc} P$ is self-inhibiting (once it has been separated into an independent loop for serialisation, it will execute at most once due to Law 5.1.12). Provided we can also ensure b holds, we can use this information to ensure P is executed exactly once.

Law 5.1.15. *Provided* $\neg(b \wedge c)$, $inh(b \rightarrow_{snc} P, b \rightarrow_{snc} P)$ *and* $inh(c \rightarrow_{snc} Q, b \rightarrow_{snc} P)$ *we have:*

$$b_S^\top; *((b \rightarrow_{snc} P) \parallel_{\hat{M}} (c \rightarrow_{snc} Q)) = b_S^\top; P; *(c \rightarrow_{snc} Q)$$

The iteration of steps with the same guard distributes over parallel by merge.

Law 5.1.16. *Provided P and Q take one clock cycle to terminate we have:*

$$*((b \underset{snc}{\rightarrow} P) \parallel_{\hat{M}} (b \underset{snc}{\rightarrow} Q)) = *(b \underset{snc}{\rightarrow} P) \parallel_{\hat{M}} *(b \underset{snc}{\rightarrow} Q)$$

5.1.3 First Normal Form

The goal of this section is to define the first normal form for our compiler and to show how to reduce all the primitive constructs in our subset of Handel-C to it.

In the reasoning language defined in previous sections, a program of the form:

$$\texttt{var}\ a; s_S^{\top}; b * P; f_S^{\perp}; \texttt{end}\ a$$

can be interpreted as the description of a state machine where:

- The variable a represents the control variables governing the execution of the machine the normal form represents. The control variables in a do not exist at the source level and are relevant to this particular normal form; hence they are introduced as local variables with scope to the normal form only.

- s is an assumption about the control state in which the normal form is started. In general, s requires a to have a particular initial value. If the machine is started in a state not satisfying s, then it behaves miraculously (freeing the implementation from dealing with such cases).

- $b * P$ expresses the behaviour of the program represented by the normal form. In general, P is expressed as the guarded choice $b_1 \underset{\text{snc}}{\rightarrow} P_1 \parallel_{\hat{M}} \ldots \parallel_{\hat{M}} b_n \underset{\text{snc}}{\rightarrow} P_n$. In this case, the form $b * P$ is achieved by having $b = b_1 \vee \cdots \vee b_n$.

  We require each P_i to be of the form $(v \underset{\text{snc}}{:=} e)_1$ for an appropriate list of variables v and corresponding list of expressions e. We also constrain P_i to be *independent* from the value of a. In practical terms, a-independence implies that P_i's behaviour does not depend on the value of a but that it assigns a new value to it. In algebraic terms, the independence from the value of a can be characterised by the fix-point law:

  $$(a \underset{\text{snc}}{:=} e; P) = P$$

  Note that we do not impose this restriction to the conditions guarding each P_i. In fact, all b_is in our compilation into normal form will be conditions depending on the control values held in a.

- f captures the value the machine's control variables should hold at the end of the execution of $b * P$. If the execution of $b * P$ failed to establish f the machine behaves like the worst possible process $\perp$.

Note that when $b * P$ has terminated, the condition b no longer holds so we have

$$b * P; f_S^{\perp} = b * P; \neg b_S^{\perp}; f_S^{\perp} = b * P; (\neg b \wedge f)_S^{\perp}$$

allowing us to assume $f = (\neg b \wedge f)$ and, consequently, $\neg(b \wedge f)$. This fact is used in some of the compilation theorems (e.g., Lemma 5.1.18) to show the semantic correspondence between the programming constructs and their normal form representations.

In order to keep the presentation compact and facilitate the manipulation of programs we will abbreviate our normal form by means of the following notation:

Definition 5.1.17. *Normal form*

$$a : [s, (b \underset{snc}{\rightarrow} P), f] =_{df} \texttt{var}\ a; s_S^{\top}; *b \underset{snc}{\rightarrow} P; f_S^{\perp}; \texttt{end}\ a$$

Taking advantage of the knowledge that P is comprised of steps of the form

$$b_1 \underset{\text{snc}}{\rightarrow} P_1 \parallel_{\hat{M}} \ldots \parallel_{\hat{M}} b_n \underset{\text{snc}}{\rightarrow} P_n$$

and apply definitions 5.1.2 and 5.1.8 to write:

$$c : [a, \left(b_1 \underset{\text{snc}}{\rightarrow} P_1 \parallel_{\hat{M}} \ldots \parallel_{\hat{M}} b_n \underset{\text{snc}}{\rightarrow} P_n\right), f]$$

as a syntactic shorthand for:

$$c : [a, \left((b_1 \vee \cdots \vee b_n) \underset{\text{snc}}{\rightarrow} \left(b_1 \underset{\text{snc}}{\rightarrow} P_1 \parallel_{\hat{M}} \ldots \parallel_{\hat{M}} b_n \underset{\text{snc}}{\rightarrow} P_n\right)\right), f]$$

5.1.4 Normal form reduction theorems

We have already introduced the definition of the state machine we will use as our first normal form together with the kind of actions it can perform (i.e., the machine steps). We will now concentrate on defining a complete set of compilation rules allowing us to reduce any program constructed from our subset of Handel-C into the first normal form.

Following the ideas used by Sampaio [1997]; Iyoda and He [2001*b*]; Duran et al. [2001]; He [2002], it is possible to prove that an arbitrary program can be reduced to normal form by means of proceeding by structural induction over the input language's constructs. In particular, it is sufficient to show how each primitive command can be written in normal form and that all the operators in the language (when applied to operands in normal form) yield a result expressible in normal form.

Even though some of the theorems in this section state straightforward results, they have this status because they show how the source language operators can be reduced to normal form. Furthermore, the theorems in this section express transformations that are architecture independent in the sense that there is no direct mapping into hardware (i.e. FPGA) elements at this level of abstraction. In Section 5.2 we address this issue and introduce further transformations that generate code in a representation where hardware elements can be easily identified and derived from the normal form.

5.1.4.1 Skip, assignment and delay

If the initial state of a machine is equal to its final state, the machine does not perform any action. In the context of our normal form, a machine that is started in its final state has, by definition, terminated already and will just ignore the set of actions inside its execution loop. The following lemma takes advantage of this observation to provide a normal form representation for the skip construct.

Lemma 5.1.18. *Normal form encoding of* II

$$\mathit{II} \sqsubseteq a : [a = s, (b \xrightarrow[snc]{} P), a = s]$$

for any valid step $b \xrightarrow[snc]{} P$

Proof.

$$\begin{array}{ll} & a : [a = s, (b \xrightarrow[snc]{} P), a = s] \\ = & \{\text{Definition 5.1.17}\} \\ & \mathtt{var}\ a; (a = s)^{\top}_{\mathsf{S}}; *(b \xrightarrow[snc]{} P); (a = s)^{\bot}_{\mathsf{S}}; \mathsf{end}\ a \\ = & \{\text{Law 3.3.98 (remember that } \neg(b \wedge f) \text{ in the normal form, then } (a = s) \Rightarrow \neg b)\} \\ & \mathtt{var}\ x; (a = s)^{\top}_{\mathsf{S}}; (a = s)^{\bot}_{\mathsf{S}}; \mathsf{end}\ a \\ \sqsupseteq & \{\text{Laws 3.3.43 and 3.3.70}\} \\ & \mathit{II} \end{array}$$

□

In order to encode the **delay** construct in normal form we need a way of stating a machine that takes one clock cycle to terminate and leaves the program variables unchanged (i.e., **delay** is just an empty multiple assignment to all the program variables). From the point of view of our normal form, this effect can be achieved by a machine with a single step that (a) keeps the program variables constant; (b) sets the control state to the machine's final value; and (c) takes a whole clock cycle to perform those updates.

Theorem 5.1.19. ***delay*** $\sqsubseteq a : [a = s, (a = s \xrightarrow[snc]{} (v, a \mathrel{:=}_{snc} v, f)_1), a = f]$

Proof.

$$\begin{array}{ll} & a : [a = s, (a = s \xrightarrow[snc]{} (v, a \mathrel{:=}_{snc} v, f)_1), a = f] \\ = & \{\text{Definition 5.1.17}\} \\ & \mathtt{var}\ a; (a = s)^{\top}_{\mathsf{S}}; *(a = s \xrightarrow[snc]{} (v, a \mathrel{:=}_{snc} v, f)_1); (a = f)^{\bot}_{\mathsf{S}}; \mathsf{end}\ a \\ = & \{\text{Laws 3.3.100 and 4.2.6}\} \\ & \mathtt{var}\ a; (a = s)^{\top}_{\mathsf{S}}; (v, a \mathrel{:=}_{snc} v, f)_1; (a = f)^{\bot}_{\mathsf{S}}; *(a = s \xrightarrow[snc]{} (v, a \mathrel{:=}_{snc} v, f)_1); (a = f)^{\bot}_{\mathsf{S}}; \mathsf{end}\ a \\ = & \{\text{Laws 3.3.98 (by definition, } s \neq f) \text{ and 4.2.6(twice)}\} \\ & \mathtt{var}\ a; (a = s)^{\top}_{\mathsf{S}}; (v, a \mathrel{:=}_{snc} v, f)_1; \mathsf{end}\ a \end{array}$$

$\sqsupseteq$ {Laws 3.3.72 and 3.3.95}

$\texttt{var}\ a; (a = s)^{\top}_{\mathsf{S}}; \texttt{end}\ a; (v \mathrel{:=}_{\mathrm{snc}} v)_1$

= {Laws 3.3.74, 3.3.70 and 3.3.9}

$(v \mathrel{:=}_{\mathrm{snc}} v)_1$

= {Definition 2.3.8 and theorem 4.2.2}

delay □

In a similar way, the assignment construct uses the same structure used in the previous theorem to encode **delay** in normal form but updating the appropriate list of variables.

Theorem 5.1.20. $x \mathrel{:=}_{HC} e \sqsubseteq a : [a = s, (a = s \overrightarrow{\ }_{snc} (x, a \mathrel{:=}_{snc} e, f)_1), a = f]$

Proof.

Similar to theorem 5.1.19 with $(x \mathrel{:=}_{\mathrm{snc}} e)_1$ instead of $(v \mathrel{:=}_{\mathrm{snc}} v)_1$. □

Having defined a way to express the basic Handel-C constructs in normal form, we turn our attention into showing how the programming operators can combine normal forms together and produce results that are in normal form.

5.1.4.2 Sequential composition

Our strategy to reduce sequential composition into normal form follows the one used by Hoare, He and Sampaio [Hoare et al. 1993] in the context of the compilation of imperative programs. The reduction of the sequential composition $P; Q$ assumes both arguments are in normal form, they share the same control variables and that the final state of P coincides with the initial state of Q. The execution loop of the resulting normal form comprises the sets of steps of P and Q.

We first consider the compilation of a particular case of sequential composition where the set of steps in the normal form on the right includes that of the one on the left.

Lemma 5.1.21. *Simple sequential composition*

$$a : [s, (b_1 \overrightarrow{\ }_{snc} P), f_0] \ ; \ a : [f_0, \begin{pmatrix} b_1 \overrightarrow{\ }_{snc} P \\ \|_{\hat{M}} \ b_2 \overrightarrow{\ }_{snc} Q \end{pmatrix}, f] \sqsubseteq a : [s, \begin{pmatrix} b_1 \overrightarrow{\ }_{snc} P \\ \|_{\hat{M}} \ b_2 \overrightarrow{\ }_{snc} Q \end{pmatrix}, f]$$

Proof.

$a : [s, (b_1 \overrightarrow{\ }_{\mathrm{snc}} P), f_0]; a : [f_0, \left(b_1 \overrightarrow{\ }_{\mathrm{snc}} P \|_{\hat{M}} \ b_2 \overrightarrow{\ }_{\mathrm{snc}} Q\right), f]$

= {Definition 5.1.17}

$\texttt{var}\ a; (a = s)^{\top}_{\mathsf{S}}; *(b_1 \overrightarrow{\ }_{\mathrm{snc}} P); (a = f_0)^{\bot}_{\mathsf{S}}; \texttt{end}\ a;$

$\texttt{var}\ a; (a = f_0)^{\top}_{\mathsf{S}} * \left(b_1 \overrightarrow{\ }_{\mathrm{snc}} P \|_{\hat{M}} \ b_2 \overrightarrow{\ }_{\mathrm{snc}} Q\right); (a = f)^{\bot}_{\mathsf{S}}; \texttt{end}\ a$

$\sqsubseteq$ {Laws 3.3.71 and 3.3.43}

$\texttt{var}\ a; (a = s)^{\top}_{\mathsf{S}}; *(b_1 \xrightarrow[\text{snc}]{} P); *\left(b_1 \xrightarrow[\text{snc}]{} P \parallel_{\hat{M}} b_2 \xrightarrow[\text{snc}]{} Q\right); (a = f)^{\perp}_{\mathsf{S}}; \mathsf{end}\ a$

= {Law 5.1.11}

$\texttt{var}\ a; (a = s)^{\top}_{\mathsf{S}}; b_1 * (b_1 \xrightarrow[\text{snc}]{} P \parallel_{\hat{M}} b_2 \xrightarrow[\text{snc}]{} Q); *\left(b_1 \xrightarrow[\text{snc}]{} P \parallel_{\hat{M}} b_2 \xrightarrow[\text{snc}]{} Q\right); (a = f)^{\perp}_{\mathsf{S}}; \mathsf{end}\ a$

= {Law 3.3.104}

$\texttt{var}\ a; (a = s)^{\top}_{\mathsf{S}}; *\left(b_1 \xrightarrow[\text{snc}]{} P \parallel_{\hat{M}} b_2 \xrightarrow[\text{snc}]{} Q\right); (a = f)^{\perp}_{\mathsf{S}}; \mathsf{end}\ a$

= {Definition 5.1.17}

$a : [s, \left(b_1 \xrightarrow[\text{snc}]{} P \parallel_{\hat{M}} b_2 \xrightarrow[\text{snc}]{} Q\right), f]$ □

Extending the set of steps in the execution loop of a normal form leads to refinement.

Lemma 5.1.22. *Normal form guarded approximation*

$$a : [s, (b_1 \xrightarrow[snc]{} P), f] \sqsubseteq a : [s, \left(b_1 \xrightarrow[snc]{} P \parallel_{\hat{M}} b_2 \xrightarrow[snc]{} Q\right), f]$$

Proof.

$a : [s, \left(b_1 \xrightarrow[\text{snc}]{} P \parallel_{\hat{M}} b_2 \xrightarrow[\text{snc}]{} Q\right), f]$

$\sqsupseteq$ {Lemma 5.1.21}

$a : [s, (b_1 \xrightarrow[\text{snc}]{} P), f]; a : [f, \left(b_1 \xrightarrow[\text{snc}]{} P \parallel_{\hat{M}} b_2 \xrightarrow[\text{snc}]{} Q\right), f]$

$\sqsupseteq$ {Lemma 5.1.18 and law 3.3.10}

$a : [s, (b_1 \xrightarrow[\text{snc}]{} P), f]$ □

The reduction theorem for the general case of the sequential composition of two normal forms is a direct consequence of the two lemmas presented above.

Theorem 5.1.23. *Sequential composition*

$$a : [s, (b_1 \xrightarrow[snc]{} P), f_1]; a : [f_1, (b_2 \xrightarrow[snc]{} Q), f] \sqsubseteq a : [s, \left(b_1 \xrightarrow[snc]{} P \parallel_{\hat{M}} b_2 \xrightarrow[snc]{} Q\right), f]$$

Proof.

$a : [s, \left(b_1 \xrightarrow[\text{snc}]{} P \parallel_{\hat{M}} b_2 \xrightarrow[\text{snc}]{} Q\right), f]$

$\sqsupseteq$ {Lemma 5.1.21}

$a : [s, (b_1 \xrightarrow[\text{snc}]{} P), f_1]; a : [f_1, \left(b_1 \xrightarrow[\text{snc}]{} P \parallel_{\hat{M}} b_2 \xrightarrow[\text{snc}]{} Q\right), f]$

$\sqsupseteq$ {Lemma 5.1.22}

$a : [s, (b_1 \xrightarrow[\text{snc}]{} P), f_1]; a : [f_1, (b_2 \xrightarrow[\text{snc}]{} Q), f]$ □

5.1.4.3 The conditional construct

Handel-C semantics requires the evaluation of the condition and the activation of the first action in the selected branch within the fist clock cycle of the conditional construct's execution.

In the context of our normal form, changes in the control flow of the program are encoded by changing the control state of the state machine (i.e., by means of updating the values in the state variables). Unfortunately, updating the control variables take a whole clock cycle (consider the fact that variables are stored in flip-flops that take a whole clock cycle to reach a stable state) and this will violate the semantics of the conditional construct.

A possible way round this problem is to require that P and Q use the same set of control variables and have the same condition guarding the first step. In this way we have that

$$P = a : [s, \left(s \overrightarrow{\ }_{\text{snc}} P_1 \parallel_{\hat{M}} b_1 \overrightarrow{\ }_{\text{snc}} P_2\right), f]$$

and

$$Q = a : [s, \left(s \overrightarrow{\ }_{\text{snc}} Q_1 \parallel_{\hat{M}} b_2 \overrightarrow{\ }_{\text{snc}} Q_2\right), f]$$

The compilation of a program of the form $(P \lhd b \rhd Q)$ into normal form could then be formulated by means of performing the selection between P_1 and Q_1 within the actions of the first step. This idea leads to a compilation schema of the form:

$$P \lhd b \rhd Q \sqsubseteq a : [s, \left(s \overrightarrow{\ }_{\text{snc}} P_1 \lhd b \rhd Q_1 \parallel_{\hat{M}} b_1 \overrightarrow{\ }_{\text{snc}} P_2 \parallel_{\hat{M}} b_2 \overrightarrow{\ }_{\text{snc}} Q_2\right), f]$$

Even though this compilation schema achieves the desired effect in terms of the timing restrictions of the conditional construct, it opens the possibility of erroneous behaviour if either of the branches ever returns to the initial state during its execution. For example consider the case where P is a simple implementation of the program **while** c **do** $(x \overset{:=}{}_{\text{HC}} x + 1)$:

$$P = a : [s, \left(s \overrightarrow{\ }_{\text{snc}} (x, a \overset{:=}{}_{\text{snc}} (x + 1, s) \lhd c \rhd (x, f))_1\right), f]$$

then, if we are to compile the program $(P \lhd x = 0 \rhd Q)$ using the approach described above we have:

$$P \lhd b \rhd Q \sqsubseteq a : [s, \left(\begin{array}{l} s \overrightarrow{\ }_{\text{snc}} \left((x, a \overset{:=}{}_{\text{snc}} (x + 1, s) \lhd c \rhd (x, f))_1 \lhd x = 0 \rhd Q_1\right) \\ \parallel_{\hat{M}} b_2 \overrightarrow{\ }_{\text{snc}} Q_2 \end{array}\right), f]$$

In the case of this program running in a context where $x = 0$, the left branch associated with P will be selected for execution and it will establish $a = s$ and $x = 1$. In the next iteration, $x = 0$ will not hold and Q_1 will be selected for execution. In this way, our implementation of the conditional construct initially selected P for execution but switched to executing Q after one clock cycle!

This problem arises because we are pushing the resolution of the conditional into the first step of a possibly re-entrant state machine. The solution is to ensure that P and Q have an initial state that is unreachable after it has been executed. The next result shows that we can add a new (and

unique) entry point to a normal form and that this addition refines the initial normal form:

Lemma 5.1.24. *Entry point state addition*

$$a : [(a = s_1), \begin{pmatrix} (a = s_1) \xrightarrow[snc]{} P_1 \\ \|_{\hat{M}}\ b \xrightarrow[snc]{} P \end{pmatrix}, f] \sqsubseteq a : [a = s, \left(a = s \xrightarrow[snc]{} P_1 \|_{\hat{M}} (a = s_1) \xrightarrow[snc]{} P_1 \|_{\hat{M}} b \xrightarrow[snc]{} P\right), f]$$

provided s is a fresh control state

Proof.

Let $S =_{df} (a = s_1) \xrightarrow[\text{snc}]{} P_1 \|_{\hat{M}} b \xrightarrow[\text{snc}]{} P$ in:

$$\begin{aligned}
& a : [s_1, \left((a = s_1) \xrightarrow[\text{snc}]{} P_1 \|_{\hat{M}} b \xrightarrow[\text{snc}]{} P\right), f] \\
= & \ \{\text{Definition 5.1.17}\} \\
& \mathtt{var}\ a; ((a = s_1))^{\top}_{\mathsf{S}}; * \left((a = s_1) \xrightarrow[\text{snc}]{} P_1 \|_{\hat{M}} b \xrightarrow[\text{snc}]{} P\right); f^{\perp}_{\mathsf{S}}; \mathsf{end}\ a \\
= & \ \{\text{Definition 5.1.8, then laws 3.3.100, 3.3.58}\} \\
& \mathtt{var}\ a; ((a = s_1))^{\top}_{\mathsf{S}}; S; * \left((a = s_1) \xrightarrow[\text{snc}]{} P_1 \|_{\hat{M}} b \xrightarrow[\text{snc}]{} P\right); f^{\perp}_{\mathsf{S}}; \mathsf{end}\ a \\
= & \ \{\text{Laws 3.3.74 and 3.3.16}\} \\
& \mathtt{var}\ a; S[s_1/a]; *P; f^{\perp}_{\mathsf{S}}; \mathsf{end}\ a \\
= & \ \{S \text{ is a step} \rightarrow S \text{ is } a\text{-independent, law 3.3.23}\} \\
& \mathtt{var}\ a; (S \lhd a = s \rhd S); * \left((a = s_1) \xrightarrow[\text{snc}]{} P_1 \|_{\hat{M}} b \xrightarrow[\text{snc}]{} P\right); f^{\perp}_{\mathsf{S}}; \mathsf{end}\ a \\
\sqsubseteq & \ \{\text{Laws 3.3.4, 3.3.7, 3.3.9 and 3.3.33}\} \\
& \mathtt{var}\ a; (\text{II} \lhd a = s \rhd \top); S; * \left((a = s_1) \xrightarrow[\text{snc}]{} P_1 \|_{\hat{M}} b \xrightarrow[\text{snc}]{} P\right); f^{\perp}_{\mathsf{S}}; \mathsf{end}\ a \\
= & \ \{\text{Definition 3.3.34}\} \\
& \mathtt{var}\ a; a = s^{\top}_{\mathsf{S}}; S; *P; f^{\perp}_{\mathsf{S}}; \mathsf{end}\ a \\
= & \ \{inh\,(a = s \xrightarrow[\text{snc}]{} S, a = s \xrightarrow[\text{snc}]{} S),\ inh\,(P, a = s \xrightarrow[\text{snc}]{} S), \text{law 5.1.15}\} \\
& \mathtt{var}\ a; a = s^{\top}_{\mathsf{S}}; * \left(a = s \xrightarrow[\text{snc}]{} S \|_{\hat{M}} (a = s_1) \xrightarrow[\text{snc}]{} P_1 \|_{\hat{M}} b \xrightarrow[\text{snc}]{} P\right); f^{\perp}_{\mathsf{S}}; \mathsf{end}\ a \\
= & \ \{\text{Definition 5.1.17}\} \\
& a : [a = s, \left(a = s \xrightarrow[\text{snc}]{} P_1 \|_{\hat{M}} (a = s_1) \xrightarrow[\text{snc}]{} P_1 \|_{\hat{M}} b \xrightarrow[\text{snc}]{} P\right), f] \qquad \square
\end{aligned}$$

The next lemma allows us to simplify a normal form with a single starting point provided it depends on a condition that is known to hold.

Lemma 5.1.25. *Entry point simplification*

$$b^{\top}_{S}; a : [s, \left(s \xrightarrow[snc]{} P_1 \lhd b \rhd P_2 \|_{\hat{M}} b_2 \xrightarrow[snc]{} Q\right), f] = b^{\top}_{S}; a : [s, \left(s \xrightarrow[snc]{} P_1 \|_{\hat{M}} b_2 \xrightarrow[snc]{} Q\right), f]$$

provided $s \xrightarrow[snc]{} (P_1 \lhd b \rhd P_2)$ is self-inhibiting and $inh\,(b_2 \xrightarrow[snc]{} Q, s \xrightarrow[snc]{} (P_1 \lhd b \rhd P_2))$

Proof.

$$
\begin{aligned}
& b_{\mathsf{S}}^{\top}; a : [s, \left(s \overrightarrow{}_{\mathrm{snc}} P_1 \lhd b \rhd P_2 \parallel_{\hat{M}} b_2 \overrightarrow{}_{\mathrm{snc}} Q\right), f] \\
= & \{\text{Definition 5.1.17}\} \\
& b_{\mathsf{S}}^{\top}; \mathtt{var}\ a; a = s_{\mathsf{S}}^{\top}; * \left(s \overrightarrow{}_{\mathrm{snc}} P_1 \lhd b \rhd P_2 \parallel_{\hat{M}} b_2 \overrightarrow{}_{\mathrm{snc}} Q\right); a = f_{\mathsf{S}}^{\perp}; \mathsf{end}\ a \\
= & \{\text{Proviso and law 5.1.15}\} \\
& b_{\mathsf{S}}^{\top}; \mathtt{var}\ a; a = s_{\mathsf{S}}^{\top}; P_1 \lhd b \rhd P_2; * (b_2 \overrightarrow{}_{\mathrm{snc}} Q); a = f_{\mathsf{S}}^{\perp}; \mathsf{end}\ a \\
= & \{\text{Laws 3.3.92, 3.3.46 and 3.3.58}\} \\
& \mathtt{var}\ a; a = s_{\mathsf{S}}^{\top}; b_{\mathsf{S}}^{\top}; P_1; * (b_2 \overrightarrow{}_{\mathrm{snc}} Q); a = f_{\mathsf{S}}^{\perp}; \mathsf{end}\ a \\
= & \{\text{Laws 3.3.92 and 3.3.46, then proviso and law 5.1.15}\} \\
& b_{\mathsf{S}}^{\top}; \mathtt{var}\ a; a = s_{\mathsf{S}}^{\top}; * \left(s \overrightarrow{}_{\mathrm{snc}} P_1 \parallel_{\hat{M}} b_2 \overrightarrow{}_{\mathrm{snc}} Q\right); a = f_{\mathsf{S}}^{\perp}; \mathsf{end}\ a \\
= & \{\text{Definition 5.1.17}\} \\
& b_{\mathsf{S}}^{\top}; a : [s, \left(s \overrightarrow{}_{\mathrm{snc}} P_1 \parallel_{\hat{M}} b_2 \overrightarrow{}_{\mathrm{snc}} Q\right), f]
\end{aligned}
$$

□

The compilation of $(P \lhd b \rhd Q)$ into first normal form is the result of firstly extending P and Q to make them single-entry by means of Lemma 5.1.24 and then applying the technique described at the beginning of this section to push the resolution of the condition inside the first step.

Theorem 5.1.26. *Conditional*

$$
\begin{aligned}
& a : [s_1, \left(s_1 \overrightarrow{}_{snc} P_1 \parallel_{\hat{M}} b_1 \overrightarrow{}_{snc} Q_1\right), f] \lhd b \rhd a : [s_2, \left(s_2 \overrightarrow{}_{snc} P_2 \parallel_{\hat{M}} b_2 \overrightarrow{}_{snc} Q_2\right), f] \sqsubseteq \\
& \quad a : [s, \left(s \overrightarrow{}_{snc} (P_1 \lhd b \rhd P_2) \parallel_{\hat{M}} \left(\begin{array}{c} s_1 \overrightarrow{}_{snc} P_1 \parallel_{\hat{M}} b_1 \overrightarrow{}_{snc} Q_1 \\ \parallel_{\hat{M}} \ s_2 \overrightarrow{}_{snc} P_2 \parallel_{\hat{M}} b_2 \overrightarrow{}_{snc} Q_2 \end{array}\right)\right), f]
\end{aligned}
$$

provided s is a fresh control state and that the encodings of P and Q in normal form are single-entry.

Proof.

We first observe that as $\neg(s \wedge (s_1 \vee b_1 \vee s_2 \vee b_2))$ (we have selected s to be a fresh control state), the first step of the normal form is unreachable once it has been executed. Algebraically,

$$inh\left(s_1 \overrightarrow{}_{\mathrm{snc}} P_1 \parallel_{\hat{M}} b_1 \overrightarrow{}_{\mathrm{snc}} Q_1, s \overrightarrow{}_{\mathrm{snc}} P\right)$$

and

$$inh\left(s_2 \overrightarrow{}_{\mathrm{snc}} P_2 \parallel_{\hat{M}} b_2 \overrightarrow{}_{\mathrm{snc}} Q_2, s \overrightarrow{}_{\mathrm{snc}} P\right)$$

for any process P. Furthermore, as P_1 and P_2 cannot mention the control state s (again, we have chosen it to satisfy this condition) then $s \overrightarrow{}_{\mathrm{snc}} (P_1 \lhd b \rhd P_2)$ is self inhibiting.

$$
\begin{aligned}
& a : [s, \left(s \overrightarrow{}_{\text{snc}} (P_1 \lhd b \rhd P_2) \parallel_{\hat{M}} \left(\begin{array}{c} s_1 \overrightarrow{}_{\text{snc}} P_1 \parallel_{\hat{M}} b_1 \overrightarrow{}_{\text{snc}} Q_1 \\ \parallel_{\hat{M}} s_2 \overrightarrow{}_{\text{snc}} P_2 \parallel_{\hat{M}} b_2 \overrightarrow{}_{\text{snc}} Q_2 \end{array} \right) \right), f] \\
= & \{\text{Law 3.3.23}\} \\
& a : [s, \left(\begin{array}{c} s \overrightarrow{}_{\text{snc}} (P_1 \lhd b \rhd P_2) \\ \parallel_{\hat{M}} s_1 \overrightarrow{}_{\text{snc}} P_1 \parallel_{\hat{M}} b_1 \overrightarrow{}_{\text{snc}} Q_1 \\ \parallel_{\hat{M}} s_2 \overrightarrow{}_{\text{snc}} P_2 \parallel_{\hat{M}} b_2 \overrightarrow{}_{\text{snc}} Q_2 \end{array} \right), f] \lhd b \rhd a : [s, \left(\begin{array}{c} s \overrightarrow{}_{\text{snc}} (P_1 \lhd b \rhd P_2) \\ \parallel_{\hat{M}} s_1 \overrightarrow{}_{\text{snc}} P_1 \parallel_{\hat{M}} b_1 \overrightarrow{}_{\text{snc}} Q_1 \\ \parallel_{\hat{M}} s_2 \overrightarrow{}_{\text{snc}} P_2 \parallel_{\hat{M}} b_2 \overrightarrow{}_{\text{snc}} Q_2 \end{array} \right), f] \\
= & \{\text{Law 3.3.57 and Lemma 5.1.25 (observation above ensures precondition)}\} \\
& a : [s, \left(\begin{array}{c} s \overrightarrow{}_{\text{snc}} P_1 \\ \parallel_{\hat{M}} s_1 \overrightarrow{}_{\text{snc}} P_1 \parallel_{\hat{M}} b_1 \overrightarrow{}_{\text{snc}} Q_1 \\ \parallel_{\hat{M}} s_2 \overrightarrow{}_{\text{snc}} P_2 \parallel_{\hat{M}} b_2 \overrightarrow{}_{\text{snc}} Q_2 \end{array} \right), f] \lhd b \rhd a : [s, \left(\begin{array}{c} s \overrightarrow{}_{\text{snc}} P_2 \\ \parallel_{\hat{M}} s_1 \overrightarrow{}_{\text{snc}} P_1 \parallel_{\hat{M}} b_1 \overrightarrow{}_{\text{snc}} Q_1 \\ \parallel_{\hat{M}} s_2 \overrightarrow{}_{\text{snc}} P_2 \parallel_{\hat{M}} b_2 \overrightarrow{}_{\text{snc}} Q_2 \end{array} \right), f] \\
\sqsupseteq & \{\text{Lemma 5.1.22}\} \\
& a : [s, \left(s \overrightarrow{}_{\text{snc}} P_1 \parallel_{\hat{M}} \left(\begin{array}{c} s_1 \overrightarrow{}_{\text{snc}} P_1 \\ \parallel_{\hat{M}} b_1 \overrightarrow{}_{\text{snc}} Q_1 \end{array} \right) \right), f] \lhd b \rhd a : [s, \left(s \overrightarrow{}_{\text{snc}} P_2 \parallel_{\hat{M}} \left(\begin{array}{c} s_2 \overrightarrow{}_{\text{snc}} P_2 \\ \parallel_{\hat{M}} b_2 \overrightarrow{}_{\text{snc}} Q_2 \end{array} \right) \right), f] \\
\sqsupseteq & \{\text{Lemma 5.1.24}\} \\
& a : [s_1, \left(s_1 \overrightarrow{}_{\text{snc}} P_1 \parallel_{\hat{M}} b_1 \overrightarrow{}_{\text{snc}} Q_1 \right), f] \lhd b \rhd a : [s_2, \left(s_2 \overrightarrow{}_{\text{snc}} P_2 \parallel_{\hat{M}} b_2 \overrightarrow{}_{\text{snc}} Q_2 \right), f] \qquad \square
\end{aligned}
$$

5.1.4.4 Iteration

When trying to provide a normal form encoding for the iteration construct, we face the problem of the immediate (i.e., combinatorially rather than in a new clock cycle) change of control state that arises when the **while** condition becomes **false** and the loop terminates. In this case, control is passed within the same clock cycle to the construct that follows the while in the sequential order imposed by the program.

This is a similar situation to the one we encountered when reducing the conditional construct to normal form. Recasting from the previous section, we solved the combinatorial change in the control flow by means of pushing conditional actions (taken from the actions that must follow the evaluation of the condition) into the same state where the condition is evaluated[1].

In the context of the iteration construct, actions to happen when the loop's condition becomes **false** are indicated by placing them in sequential composition after the while construct. This means the iteration construct does not carry any information regarding the actions that will follow its execution. This automatically pre-empts us from solving the problem with the same idea we used to reduce the selection construct to normal form.

On the other hand, if we require all loops in the program to be followed by a combination of constructs that takes at least one clock cycle we can apply Law 3.3.103 and Definition 4.2.7 to encode them as an iterating selection as defined in Section 4.2.2. The main advantage of expressing

[1]Consider, for example, the case of the selection construct (**if** c **then** P **else** Q). We can always extract the behaviour that follows from the conditional point c from P or Q depending on the value of the condition.

loops in this way is that information about the actions to be performed when the loop terminates is contained within the iterating selection construct. In this way, we can compile this particular kind of iteration using the strategy described in the previous section for the reduction of the conditional construct to normal form.

The requirement that loops are followed by at least one time-consuming action is, in general, satisfied in most programs written in Handel-C. On the other hand, it is possible to write valid programs that do not satisfy this condition by performing an iteration in the following cases:

1. just before terminating its execution:

   P; **while** b **do** Q

2. at the end of one of the parallel branches:

   (**while** b **do** Q) $\|_{\hat{M}}$ P

3. at the end of a conditional branch:

   (**while** b **do** Q) $\lhd$ c $\rhd$ P

In all these cases the user will be forced to place a time consuming action after the **while** construct if the program is to be compiled using our approach.

We have conducted a pilot study taking sample Handel-C programs as described in [Aubury et al. 1996]. The examples range from a simple accumulator to a multi-module histogram equaliser. Our pilot study does not intend to provide formal evidence that our request for **while** constructs to be followed by a time-consuming action will always be trivially satisfied or that we will always be able to transform a program automatically to meet this form. The main aim of the study was to provide some insight on how often our requirement will force the modification of the source program in order to meet our requirements.

Only one of the example programs, a parallel implementation of a 4-places queue, was found not to be in the format required by iterated selection. The code of the queue implements a server that is meant to run forever (i.e., until the hardware breaks or is disconnected) thus the code follows the structure

main(){while true do Q**}**

matching a particular form of case 1 described above. In this case, a **delay** construct can be easily added after the **while** construct and, as the new **delay** will never be executed, the semantics of the program are not changed.

From the evidence above, we decided that our compiler will only have the iterating selection construct as mechanism to implement loops. The user may need, in some cases, to change the source program to match our requirement and these changes may lead to some minor performance

losses. On the other hand, our compiler is intended to be used in the context of safety-critical systems where, in most cases, correctness is a more relevant requirement than performance.

5.1.4.5 Iterating selection

The reduction of the iterating selection construct to normal form follows a similar approach to the one presented for the selection construct by adding a new unique initial state where the looping condition is evaluated. The main difference in the case of the iterating selection is that the new state is set to be the final state of the iteration's body. In this way, we ensure the looping condition gets evaluated every time an execution of the loop's body terminates, which exactly matches the semantics of the looping part of the iterating selection.

We begin by proving a result similar to recursion unfolding for normal forms: when the iteration's condition hold, the body of the loop executes once and the loop starts again. The following lemma captures this notion by extracting the normal form corresponding to the loop's body and putting it in sequence with the complete loop. The presence of assumptions and assertions weakens the usual equality of the unfolding law into refinement.

Lemma 5.1.27.

$$b_S^\top; a : [s_1, \left(s_1 \overrightarrow{}_{snc} P_1 \parallel_{\hat{M}} b_1 \overrightarrow{}_{snc} S_1\right), f_1]; a : [f_1, \begin{pmatrix} f_1 \overrightarrow{}_{snc} P_1 \lhd b \rhd P_2 \\ \parallel_{\hat{M}} s_1 \overrightarrow{}_{snc} P_1 \parallel_{\hat{M}} b_1 \overrightarrow{}_{snc} S_1 \\ \parallel_{\hat{M}} s_2 \overrightarrow{}_{snc} P_2 \parallel_{\hat{M}} b_2 \overrightarrow{}_{snc} S_2 \end{pmatrix}, f] \sqsubseteq$$
$$b_S^\top; a : [f_1, \left(f_1 \overrightarrow{}_{snc} P_1 \lhd b \rhd P_2 \parallel_{\hat{M}} s_1 \overrightarrow{}_{snc} P_1 \parallel_{\hat{M}} b_1 \overrightarrow{}_{snc} S_1 \parallel_{\hat{M}} s_2 \overrightarrow{}_{snc} P_2 \parallel_{\hat{M}} b_2 \overrightarrow{}_{snc} S_2\right), f]$$

Proof.

Let $P = f_1 \overrightarrow{}_{snc} P_1 \lhd b \rhd P_2 \parallel_{\hat{M}} s_1 \overrightarrow{}_{snc} P_1 \parallel_{\hat{M}} b_1 \overrightarrow{}_{snc} S_1 \parallel_{\hat{M}} s_2 \overrightarrow{}_{snc} P_2 \parallel_{\hat{M}} b_2 \overrightarrow{}_{snc} S_2$ in:

$b_S^\top; a : [s_1, \left(s_1 \overrightarrow{}_{snc} P_1 \parallel_{\hat{M}} b_1 \overrightarrow{}_{snc} S_1\right), f_1]; a : [f_1, P, f]$

= {Definition 5.1.17, then law 3.3.100}

$b_S^\top; \mathtt{var}\ a; (a = s_1)_S^\top; P_1; *(s_1 \overrightarrow{}_{snc} P_1 \parallel_{\hat{M}} b_1 \overrightarrow{}_{snc} S_1); (a = f_1)_S^\perp; \mathsf{end}\ a;$

$\mathtt{var}\ a; (a = f_1)_S^\top; *P; (a = f)_S^\perp; \mathsf{end}\ a$

$\sqsubseteq$ {Laws 3.3.71, 3.3.44 and 3.3.74}

$b_S^\top; \mathtt{var}\ a; a \overset{:=}{}_{snc} s_1; P_1; *(s_1 \overrightarrow{}_{snc} P_1 \parallel_{\hat{M}} b_1 \overrightarrow{}_{snc} S_1); *P; (a = f)_S^\perp; \mathsf{end}\ a$

= {P_1 is a-independent and it performs an assignment to a, laws 3.3.16 and 3.3.92}

$\mathtt{var}\ a; b_S^\top; P_1; *(s_1 \overrightarrow{}_{snc} P_1 \parallel_{\hat{M}} b_1 \overrightarrow{}_{snc} S_1); *P; (a = f)_S^\perp; \mathsf{end}\ a$

$\sqsubseteq$ {Law 3.3.58, then laws 3.3.92, 3.3.40 and 5.1.11}

$b_S^\top; \mathtt{var}\ a; (a = f_1)_S^\top; (P_1 \lhd b \rhd P_2); (s_1 \vee b_1) * P; *P; (a = f)_S^\perp; \mathsf{end}\ a$

= {Laws 3.3.104 and 3.3.100}

$b_S^\top; \mathtt{var}\ a; (a = f_1)_S^\top; *P; (a = f)_S^\perp; \mathsf{end}\ a$

= {Definition 5.1.17}

$b^{\top}_{\mathrm{S}}; a : [f_1, \left(f_1 \underset{\mathrm{snc}}{\rightarrow} P_1 \lhd b \rhd P_2 \parallel_{\hat{M}} s_1 \underset{\mathrm{snc}}{\rightarrow} P_1 \parallel_{\hat{M}} b_1 \underset{\mathrm{snc}}{\rightarrow} S_1 \parallel_{\hat{M}} s_2 \underset{\mathrm{snc}}{\rightarrow} P_2 \parallel_{\hat{M}} b_2 \underset{\mathrm{snc}}{\rightarrow} S_2\right), f]$ □

On the other hand, when the looping condition does not hold, the iterating selection construct selects the *else* branch for execution. As with Lemma 5.1.27, the following result can be seen as the normal form equivalent of the unfolding law (in the case where the looping condition does not hold).

Lemma 5.1.28.

$$(\neg b)^{\top}_{\mathrm{S}}; a : [s_2, \left(s_2 \underset{\mathrm{snc}}{\rightarrow} P_2 \parallel_{\hat{M}} b_2 \underset{\mathrm{snc}}{\rightarrow} S_2\right), f] \sqsubseteq (\neg b)^{\top}_{\mathrm{S}}; a : [f_1, \begin{pmatrix} f_1 \underset{\mathrm{snc}}{\rightarrow} P_1 \lhd b \rhd P_2 \\ \parallel_{\hat{M}} s_1 \underset{\mathrm{snc}}{\rightarrow} P_1 \parallel_{\hat{M}} b_1 \underset{\mathrm{snc}}{\rightarrow} S_1 \\ \parallel_{\hat{M}} s_2 \underset{\mathrm{snc}}{\rightarrow} P_2 \parallel_{\hat{M}} b_2 \underset{\mathrm{snc}}{\rightarrow} S_2 \end{pmatrix}, f]$$

Proof.

Let $P = f_1 \underset{\mathrm{snc}}{\rightarrow} P_1 \lhd b \rhd P_2 \parallel_{\hat{M}} s_1 \underset{\mathrm{snc}}{\rightarrow} P_1 \parallel_{\hat{M}} b_1 \underset{\mathrm{snc}}{\rightarrow} S_1 \parallel_{\hat{M}} s_2 \underset{\mathrm{snc}}{\rightarrow} P_2 \parallel_{\hat{M}} b_2 \underset{\mathrm{snc}}{\rightarrow} S_2$ in:

$(\neg b)^{\top}_{\mathrm{S}}; a : [s_2, \left(s_2 \underset{\mathrm{snc}}{\rightarrow} P_2 \parallel_{\hat{M}} b_2 \underset{\mathrm{snc}}{\rightarrow} S_2\right), f]$

⊑ {Lemma 5.1.22, definition 5.1.17, then law 3.3.100}

$(\neg b)^{\top}_{\mathrm{S}}; \mathtt{var}\ a; (a = s_2)^{\top}_{\mathrm{S}}; P_2; *P; (a = f)^{\perp}_{\mathrm{S}}; \mathsf{end}\ a$

= {Law 3.3.74, P_1 is a-independent, 3.3.92}

$\mathtt{var}\ a; (\neg b)^{\top}_{\mathrm{S}}; P_2; *P; (a = f)^{\perp}_{\mathrm{S}}; \mathsf{end}\ a$

⊑ {Law 3.3.58, then laws 3.3.92 and 3.3.40}

$(\neg b)^{\top}_{\mathrm{S}}; \mathtt{var}\ a; (a = f_1)^{\top}_{\mathrm{S}}; (P_1 \lhd b \rhd P_2); *P; (a = f)^{\perp}_{\mathrm{S}}; \mathsf{end}\ a$

= {Law 3.3.100 and definition 5.1.17}

$(\neg b)^{\top}_{\mathrm{S}}; a : [f_1, \left(f_1 \underset{\mathrm{snc}}{\rightarrow} P_1 \lhd b \rhd P_2 \parallel_{\hat{M}} s_1 \underset{\mathrm{snc}}{\rightarrow} P_1 \parallel_{\hat{M}} b_1 \underset{\mathrm{snc}}{\rightarrow} S_1 \parallel_{\hat{M}} s_2 \underset{\mathrm{snc}}{\rightarrow} P_2 \parallel_{\hat{M}} b_2 \underset{\mathrm{snc}}{\rightarrow} S_2\right), f]$ □

The compilation of the iterating selection into first normal form is a direct consequence of the two lemmas described above and the least fix-point law.

Theorem 5.1.29.

$$b * \left(a : [s_1, \left(s_1 \underset{\mathrm{snc}}{\rightarrow} P_1 \parallel_{\hat{M}} b_1 \underset{\mathrm{snc}}{\rightarrow} S_1\right), f_1]\right) \blacktriangleright a : [s_2, \left(s_2 \underset{\mathrm{snc}}{\rightarrow} P_2 \parallel_{\hat{M}} b_2 \underset{\mathrm{snc}}{\rightarrow} S_2\right), f] \sqsubseteq$$
$$a : [f_1, \begin{pmatrix} f_1 \underset{\mathrm{snc}}{\rightarrow} P_1 \lhd b \rhd P_2 \\ \parallel_{\hat{M}} s_1 \underset{\mathrm{snc}}{\rightarrow} P_1 \parallel_{\hat{M}} b_1 \underset{\mathrm{snc}}{\rightarrow} S_1 \\ \parallel_{\hat{M}} s_2 \underset{\mathrm{snc}}{\rightarrow} P_2 \parallel_{\hat{M}} b_2 \underset{\mathrm{snc}}{\rightarrow} S_2 \end{pmatrix}, f]$$

Proof.

$$
\begin{array}{l}
\quad RHS = RHS \\
\equiv \{\text{Laws 3.3.23 and 3.3.57}\} \\
\quad RHS = (b^{\top}_{\mathrm{S}}; RHS) \lhd b \rhd ((\neg b)^{\top}_{\mathrm{S}}; RHS) \\
\equiv \{\text{Lemmas 5.1.27 and 5.1.28, law 3.3.57}\} \\
\quad RHS \sqsupseteq (a : [s_1, \left(s_1 \overrightarrow{\ }_{\mathrm{snc}} P_1 \parallel_{\hat{M}} b_1 \overrightarrow{\ }_{\mathrm{snc}} S_1\right), f_1]; RHS) \lhd b \rhd \\
\qquad\qquad a : [s_2, \left(s_2 \overrightarrow{\ }_{\mathrm{snc}} P_2 \parallel_{\hat{M}} b_2 \overrightarrow{\ }_{\mathrm{snc}} S_2\right), f] \\
\Rightarrow \{\text{Law 2.3.81}\} \\
\quad RHS \sqsupseteq \mu X \bullet ((a : [s_1, \left(s_1 \overrightarrow{\ }_{\mathrm{snc}} P_1 \parallel_{\hat{M}} b_1 \overrightarrow{\ }_{\mathrm{snc}} S_1\right), f_1]; X) \lhd b \rhd \\
\qquad\qquad a : [s_2, \left(s_2 \overrightarrow{\ }_{\mathrm{snc}} P_2 \parallel_{\hat{M}} b_2 \overrightarrow{\ }_{\mathrm{snc}} S_2\right), f]) \\
\equiv \{\text{Theorem 4.2.8}\} \\
\quad RHS \sqsupseteq LHS
\end{array}
$$

□

5.1.4.6 Parallel composition

The reduction to normal form of $P \parallel_{\hat{M}} Q$ (where P and Q are already in first normal form) takes advantage of the fact that the combination of steps is performed by the same operator used to combine P and Q in parallel. This is a major improvement from other approaches used to combine normal forms in parallel by means of the cross-product machine of P and Q [He et al. 1993].

The key to our parallel encoding is the fact that P and Q have independent control variables. In this way, we can avoid control-interference between the steps of the two machines when combined together in a single normal form. Unfortunately, this same independence between P and Q could potentially produce incorrect behaviour. To see why, consider the program

$$((x \overset{:=}{}_{\mathrm{HC}} e_1 \fatsemi y \overset{:=}{}_{\mathrm{HC}} e_2) \parallel_{\mathrm{HC}} z \overset{:=}{}_{\mathrm{HC}} e_2) \fatsemi (P \parallel_{\mathrm{HC}} Q)$$

In this case, the normal form reduction strategy will generate a normal form for $(x \overset{:=}{}_{\mathrm{HC}} e_1 \fatsemi y \overset{:=}{}_{\mathrm{HC}} e_2)$ depending on a control variable a_0 and another one for $(z \overset{:=}{}_{\mathrm{HC}} e_2)$ depending on a_1.

$$
\begin{array}{l}
a_0 : [a_0 = s_0, \left(a_0 = s_0 \overrightarrow{\ }_{\mathrm{snc}} (x, a_0 \overset{:=}{}_{\mathrm{snc}} e_1, s_1)_1 \parallel_{\hat{M}} a_0 = s_1 \overrightarrow{\ }_{\mathrm{snc}} (y, a_0 \overset{:=}{}_{\mathrm{snc}} e_2, s_2)_1\right), a_0 = s_2] \parallel_{\hat{M}} \\
a_1 : [a_1 = s_3, (a_1 = s_3 \overrightarrow{\ }_{\mathrm{snc}} (z, a_1 \overset{:=}{}_{\mathrm{snc}} e_3, s_4)_1), a_1 = s_4]
\end{array}
$$

As we are to re-use control variables (FPGA devices are constrained in resources thus the need to keep the size of our normal form as compact as possible), P and Q will also have control variables a_0 and a_1 respectively. Furthermore, the way in which sequential composition merges two normal forms together will make the final control values for a_0 and a_1 after executing $((x \overset{:=}{}_{\mathrm{HC}} e_1 \fatsemi y \overset{:=}{}_{\mathrm{HC}} e_2) \parallel_{\mathrm{HC}} z \overset{:=}{}_{\mathrm{HC}} e_2)$ to be the initial values for the normal form encoding of $(P \parallel_{\mathrm{HC}} Q)$. If the compilation strategy does not add any sort of control to ensure the simultaneous activation of P and Q, a_1 will be set to the value s_4 after one clock cycle and this will lead to the activation of the first step of Q

while P will only be activated one clock cycle later (due to the extra clock cycle required for a_0 to reach the value s_2).

To solve this potential problem, we ensure simultaneous initiation of all parallel branches by means of forcing a common initial step (cross-product like) between the actions in the first steps of P and Q. The state-based condition guarding this new step requires the introduction of a fresh state for both control variables. The need for fresh control states for this new initial state is based on the fact that any of the parallel branches could return to its initial state (consider, for example, the case of one of the branches being an iteration).

Theorem 5.1.30 below applies these notions in the compilation of parallel composition to normal form. The key effect of this theorem is to encode the parallel execution of the two normal forms as the parallel execution of their steps by means of Law 5.1.16. The proof of the theorem provides more details in this direction.

Theorem 5.1.30.

$$a_1 : [a_1 = s_{1.1}, (b_1 \overrightarrow{}_{snc} P_1), a_1 = f_1] \parallel_{\hat{M}} a_2 : [a_2 = s_{2.1}, (b_2 \overrightarrow{}_{snc} P_2), a_2 = f_2] \sqsubseteq$$
$$a_1, a_2 : [(a_1 = s_1 \wedge a_2 = s_2), P, (a_1 = f_1 \wedge a_2 = f_2)]$$

where:

- *s_1 and s_2 are fresh control states (i.e., they are not reachable from P_1 or P_2)*
- $(b_1 \overrightarrow{}_{snc} P_1) = (s_{1.1} \overrightarrow{}_{snc} P_{1.1} \parallel_{\hat{M}} b_1 \overrightarrow{}_{snc} P_{1.2})$
- $(b_2 \overrightarrow{}_{snc} P_2) = (s_{2.1} \overrightarrow{}_{snc} P_{2.1} \parallel_{\hat{M}} b_2 \overrightarrow{}_{snc} P_{2.2})$
- $P = \begin{pmatrix} (a_1 = s_1 \wedge a_2 = s_2) \overrightarrow{}_{snc} P_{1.1} \parallel_{\hat{M}} b_1 \overrightarrow{}_{snc} P_1 \parallel_{\hat{M}} \\ (a_1 = s_1 \wedge a_2 = s_2) \overrightarrow{}_{snc} P_{2.1} \parallel_{\hat{M}} b_2 \overrightarrow{}_{snc} P_2 \end{pmatrix}$

Proof.

Observation 1: Let $b = b_1 \vee b_2$ then we have:

$$\begin{aligned} & b \overrightarrow{}_{snc} (b_1 \overrightarrow{}_{snc} P_1) \\ = & \{b = b_1 \vee b_2 \text{ and law 3.3.152}\} \\ & (b_1 \vee b_2) \wedge b_1 \overrightarrow{}_{snc} P_1 \\ = & \{\text{Absorption of conjunct}\} \\ & b_1 \overrightarrow{}_{snc} P_1 \end{aligned}$$

Similarly, $b \overrightarrow{}_{snc} (b_2 \overrightarrow{}_{snc} P_2) = b_2 \overrightarrow{}_{snc} P_2$

Observation 2: Because $b_1 \rightarrow P_1$ is a step of our first normal form we know it takes exactly one clock cycle (remember this step belongs to one of the parallel branches already in normal form). Thus, it is easy to see that $b_1 \rightarrow P_1 \parallel_{\hat{M}} (\neg b_1 \wedge b_2) \rightarrow (c_1 \stackrel{:=}{}_{snc} f)_1$ also takes one clock cycle. Similarly, $b_2 \rightarrow P_2$ and $b_2 \rightarrow P_2 \parallel_{\hat{M}} (b_1 \wedge \neg b_2) \rightarrow (c_2 \stackrel{:=}{}_{snc} f)_1$ take exactly one clock cycle.

Observation 3: $s_1 \wedge s_2 \mathbin{\overrightarrow{\text{snc}}} P_{1.1} \parallel_{\hat{M}} s_1 \wedge s_2 \mathbin{\overrightarrow{\text{snc}}} P_{2.1}$ is self-inhibiting. The reason for this is that s_1 and s_2 are not among the possible control states $P_{1.1}$ and $P_{2.1}$ can reach (in fact, s_1 and s_2 are fresh control states we have introduced).

$$
\begin{aligned}
& a_1, a_2 : [(a_1 = s_1 \wedge a_2 = s_2), P, (a_1 = f_1 \wedge a_2 = f_2)] \\
= \; & \{\text{Definition 5.1.17, observation 1}\} \\
& \texttt{var}\ a_1, a_2; ((a_1 = s_1 \wedge a_2 = s_2))^{\top}_{\mathsf{S}}; \\
& * \begin{pmatrix} (a_1 = s_1 \wedge a_2 = s_2) \mathbin{\overrightarrow{\text{snc}}} P_{1.1} \parallel_{\hat{M}} b \mathbin{\overrightarrow{\text{snc}}} (b_1 \mathbin{\overrightarrow{\text{snc}}} P_1) \parallel_{\hat{M}} \\ (a_1 = s_1 \wedge a_2 = s_2) \mathbin{\overrightarrow{\text{snc}}} P_{2.1} \parallel_{\hat{M}} b \mathbin{\overrightarrow{\text{snc}}} (b_2 \mathbin{\overrightarrow{\text{snc}}} P_2) \end{pmatrix}; (a_1 = f \wedge a_2 = f)^{\perp}_{\mathsf{S}}; \mathsf{end}\ a_1; a_2 \\
= \; & \{\neg((a_1 = s_1 \wedge a_2 = s_2) \wedge b), \text{law 5.1.14}\} \\
& \texttt{var}\ a_1, a_2; ((a_1 = s_1 \wedge a_2 = s_2))^{\top}_{\mathsf{S}}; \\
& * ((a_1 = s_1 \wedge a_2 = s_2) \mathbin{\overrightarrow{\text{snc}}} P_{1.1} \parallel_{\hat{M}} (a_1 = s_1 \wedge a_2 = s_2) \mathbin{\overrightarrow{\text{snc}}} P_{2.1}); \\
& * (b \mathbin{\overrightarrow{\text{snc}}} (b_1 \mathbin{\overrightarrow{\text{snc}}} P_1) \parallel_{\hat{M}} b \mathbin{\overrightarrow{\text{snc}}} (b_2 \mathbin{\overrightarrow{\text{snc}}} P_2)); (a_1 = f \wedge a_2 = f)^{\perp}_{\mathsf{S}}; \mathsf{end}\ a_1; a_2 \\
= \; & \{\text{Laws 5.1.9, 3.3.143, 3.3.58 and 5.1.16, observation 1}\} \\
& \texttt{var}\ a_1, a_2; ((a_1 = s_1 \wedge a_2 = s_2))^{\top}_{\mathsf{S}}; P_{1.1} \parallel_{\hat{M}} P_{2.1}; \\
& * ((a_1 = s_1 \wedge a_2 = s_2) \mathbin{\overrightarrow{\text{snc}}} P_{1.1} \parallel_{\hat{M}} (a_1 = s_1 \wedge a_2 = s_2) \mathbin{\overrightarrow{\text{snc}}} P_{2.1}); \\
& \left(*(b_1 \mathbin{\overrightarrow{\text{snc}}} P_1) \parallel_{\hat{M}} *(b_2 \mathbin{\overrightarrow{\text{snc}}} P_2)\right); (a_1 = f_1 \wedge a_2 = f_2)^{\perp}_{\mathsf{S}}; \mathsf{end}\ a_1; a_2 \\
\sqsupseteq \; & \{\text{Law 3.3.40, observation 3 then law 5.1.12}\} \\
& \texttt{var}\ a_1, a_2; P_{1.1} \parallel_{\hat{M}} P_{2.1}; \left(*(b_1 \mathbin{\overrightarrow{\text{snc}}} P_1) \parallel_{\hat{M}} *(b_2 \mathbin{\overrightarrow{\text{snc}}} P_2)\right); (a_1 = f_1 \wedge a_2 = f_2)^{\perp}_{\mathsf{S}}; \mathsf{end}\ a_1; a_2 \\
= \; & \{\text{Neither } a_1 \text{ nor } a_2 \text{ appears in the right-hand side of } (P_{1.1} \parallel_{\hat{M}} P_{2.1}), \text{ law 3.3.16}\} \\
& \texttt{var}\ a_1, a_2; (a_1, a_2 \mathbin{\overset{:=}{\text{snc}}} s_{1.1}, s_{2.1}); P_{1.1} \parallel_{\hat{M}} P_{2.1}; \\
& \left(*(b_1 \mathbin{\overrightarrow{\text{snc}}} P_1) \parallel_{\hat{M}} *(b_2 \mathbin{\overrightarrow{\text{snc}}} P_2)\right); (a_1 = f_1 \wedge a_2 = f_2)^{\perp}_{\mathsf{S}}; \mathsf{end}\ a_1; a_2 \\
= \; & \{\text{Laws 3.3.74, 3.3.47, 3.3.144(twice)}\} \\
& \texttt{var}\ a_1, a_2; (a_1 = s_{1.1} \wedge a_2 = s_{2.1})^{\top}_{\mathsf{S}}; P_{1.1} \parallel_{\hat{M}} P_{2.1}; \\
& \left((*(b_1 \mathbin{\overrightarrow{\text{snc}}} P_1); (a_1 = f_1)^{\perp}_{\mathsf{S}}) \parallel_{\hat{M}} (*(b_2 \mathbin{\overrightarrow{\text{snc}}} P_2); (a_2 = f_2)^{\perp}_{\mathsf{S}})\right); \mathsf{end}\ a_1; a_2 \\
\sqsupseteq \; & \{\text{Laws 3.3.143, 3.3.45, 3.3.40, 3.3.142 (observation 2 ensures one } \mathsf{sync} \text{ event)}\} \\
& \texttt{var}\ a_1, a_2; \begin{pmatrix} \left((a_1 = s_{1.1})^{\top}_{\mathsf{S}}; P_{1.1}; *(b_1 \mathbin{\overrightarrow{\text{snc}}} P_1); (a_1 = f_1)^{\perp}_{\mathsf{S}}\right) \\ \parallel_{\hat{M}} \left((a_2 = s_{0.2})^{\top}_{\mathsf{S}}; P_{2.1}; *(b_2 \mathbin{\overrightarrow{\text{snc}}} P_2); (a_2 = f_2)^{\perp}_{\mathsf{S}}\right) \end{pmatrix}; \mathsf{end}\ a_1; a_2 \\
= \; & \{\text{Law 3.3.100}\} \\
& \texttt{var}\ a_1, a_2; \begin{pmatrix} \left((a_1 = s_{1.1})^{\top}_{\mathsf{S}}; *(b_1 \mathbin{\overrightarrow{\text{snc}}} P_1); (a_1 = f_1)^{\perp}_{\mathsf{S}}\right) \\ \parallel_{\hat{M}} \left((a_2 = s_{0.2})^{\top}_{\mathsf{S}}; *(b_2 \mathbin{\overrightarrow{\text{snc}}} P_2); (a_2 = f_2)^{\perp}_{\mathsf{S}}\right) \end{pmatrix}; \mathsf{end}\ a_1; a_2
\end{aligned}
$$

$\sqsupseteq$ {Law 3.3.148}

$(\texttt{var}\ a_1; (a_1 = s_{1.1})^{\top}_{\mathsf{S}}; *(b_1 \overrightarrow{\ }_{\mathsf{snc}} P_1); (a_1 = f_1)^{\bot}_{\mathsf{S}}; \texttt{end}\ a_1) \parallel_{\hat{M}}$

$(\texttt{var}\ a_2; (a_2 = s_{0.2})^{\top}_{\mathsf{S}}; *(b_2 \overrightarrow{\ }_{\mathsf{snc}} P_2); (a_2 = f_2)^{\bot}_{\mathsf{S}}; \texttt{end}\ a_2)$

= {Definition 5.1.17}

$a_1 : [a_1 = s_{1.1}, (b_1 \overrightarrow{\ }_{\mathsf{snc}} P_1), a_1 = f_1] \parallel_{\hat{M}} a_2 : [a_2 = s_{2.1}, (b_2 \overrightarrow{\ }_{\mathsf{snc}} P_2), a_2 = f_2]$ □

5.1.5 Extending the control variables of a normal form

In the previous section we assumed all normal forms will have the same number of control variables. On the other hand, our compilation strategy for the parallel construct produces a normal form with an extended set of control variables every time it is used. For example, the simple parallel program

$$(y \overset{:=}{}_{\mathsf{HC}} e_1 \parallel_{\mathsf{HC}} z \overset{:=}{}_{\mathsf{HC}} e_2)$$

compiles into the following first normal form:

$y \overset{:=}{}_{\mathsf{HC}} e_1 \parallel_{\mathsf{HC}} z \overset{:=}{}_{\mathsf{HC}} e_2$

= {Definitions 4.1.1, 4.1.6}

$y \overset{:=}{}_{\mathsf{snc}} e_1 \parallel_{\hat{M}} z \overset{:=}{}_{\mathsf{snc}} e_2$

$\sqsubseteq$ {Theorem 5.1.20, change name of control variables}

$a_1 : [a = s_{1.1}, (a_1 = s_{1.1} \overrightarrow{\ }_{\mathsf{snc}} (y, a_1 \overset{:=}{}_{\mathsf{snc}} e_1, f_1)_1), a_1 = f_1] \parallel_{\hat{M}}$

$a_2 : [a_2 = s_{2.1}, (a_2 = s_{2.1} \overrightarrow{\ }_{\mathsf{snc}} (z, a_2 \overset{:=}{}_{\mathsf{snc}} e_2, f_2)_1), a_2 = f_2]$

$\sqsubseteq$ {Theorem 5.1.30, let $s = (a_1 = s_1 \wedge a_2 = s_2)$ and $f = (a_1 = f_1 \wedge a_2 = f_2)$}

$$a_1, a_2 : [s, \begin{pmatrix} s \overrightarrow{\ }_{\mathsf{snc}} (y, a_1 \overset{:=}{}_{\mathsf{snc}} e_1, f_1)_1 \parallel_{\hat{M}} s \overrightarrow{\ }_{\mathsf{snc}} (z, a_2 \overset{:=}{}_{\mathsf{snc}} e_2, f_2)_1 \parallel_{\hat{M}} \\ a_1 = s_{1.1} \overrightarrow{\ }_{\mathsf{snc}} (y, a_1 \overset{:=}{}_{\mathsf{snc}} e_1, f_1)_1 \parallel_{\hat{M}} \\ a_2 = s_{2.1} \overrightarrow{\ }_{\mathsf{snc}} (z, a_2 \overset{:=}{}_{\mathsf{snc}} e_2, f_2)_1 \end{pmatrix}, f]$$

= {Law 3.3.153, propositional calculus, unreachable steps (laws 5.1.14, 5.1.13 and 3.3.99)}

$a_1, a_2 : [s, (s \overrightarrow{\ }_{\mathsf{snc}} (y, z, a_1, a_2 \overset{:=}{}_{\mathsf{snc}} e_1, e_2, f_1, f_2)_1), f]$

If we are now interested in the program

$$x \overset{:=}{}_{\mathsf{HC}} e_0 \fatsemi (y \overset{:=}{}_{\mathsf{HC}} e_1 \parallel_{\mathsf{HC}} z \overset{:=}{}_{\mathsf{HC}} e_2)$$

we have that:

$$x \overset{:=}{}_{\mathsf{HC}} e_0 \fatsemi (y \overset{:=}{}_{\mathsf{HC}} e_1 \parallel_{\mathsf{HC}} z \overset{:=}{}_{\mathsf{HC}} e_2)$$

$$
\begin{array}{l}
= \{\text{Definitions 4.1.1, 4.1.6}\} \\
\quad x \mathrel{:=}_{\text{snc}} e_0; (y \mathrel{:=}_{\text{snc}} e_1 \parallel_{\hat{M}} z \mathrel{:=}_{\text{snc}} e_2) \\
\sqsubseteq \{\text{Theorem 5.1.20 and observation above}\} \\
\quad a_1 : [a_1 = s_0, (a_1 = s_0 \overrightarrow{;}_{\text{snc}} (y, a_1 \mathrel{:=}_{\text{snc}} e_1, s_{1.1})_1), a_1 = s_{1.0}]; \\
\quad a_1, a_2 : [s, (s \overrightarrow{;}_{\text{snc}} (y, z, a_1, a_2 \mathrel{:=}_{\text{snc}} e_1, e_2, f_1, f_2)_1), f]
\end{array}
$$

The problem here is that our theorem for sequential composition requires both normal forms to have the same set of control variables. The solution is to extend the alphabet of the first normal form to include a_2 and to make it assume the initial value the parallel normal form is expecting on it. In general terms, the strategy consists of expanding the normal form to contain the missing control variables. The new control variables are then initialised to the value expected by the subsequent normal form (implementing the parallel construct). The strategy described above can be formally described as follows:

Theorem 5.1.31. *Control variable extension*

$$
\begin{array}{l}
a_1 : [a_1 = s_1, \left(b_1 \overrightarrow{;}_{snc} P_1 \parallel_{\hat{M}} b_2 \overrightarrow{;}_{snc} P_2\right), a_1 = f] \sqsubseteq \\
\qquad a_1, a_2 : [a_1 = s_1 \wedge a_2 = s_2, \left(b_1 \overrightarrow{;}_{snc} P_{1[a_2]} \parallel_{\hat{M}} b_2 \overrightarrow{;}_{snc} P_{2[a_2]}\right), a_1 = f \wedge a_2 = s_2]
\end{array}
$$

provided a_2 is the required additional control variable to be kept constant at value s_2.

Proof.

$$
\begin{array}{l}
\quad a_1 : [a_1 = s_1, \left(b_1 \overrightarrow{;}_{\text{snc}} P_1 \parallel_{\hat{M}} b_2 \overrightarrow{;}_{\text{snc}} P_2\right), a_1 = f] \\
= \{\text{Definition 5.1.17, laws 3.3.9 and 3.3.70}\} \\
\quad \mathtt{var}\ a_2; \mathsf{end}\ a_2; \mathtt{var}\ a_1; (a_1 = s_1)^{\top}_{\mathrm{S}}; (b_1 \vee b_2) * (b_1 \overrightarrow{;}_{\text{snc}} P_1 \parallel_{\hat{M}} b_2 \overrightarrow{;}_{\text{snc}} P_2); (a_1 = f)^{\perp}_{\mathrm{S}}; \mathsf{end}\ a_1 \\
\sqsubseteq \{\text{Laws 3.3.73, 3.3.74, 3.3.95 and 3.3.76, predicate calculus then law 3.3.102}\} \\
\quad \mathtt{var}\ a_2; (a_2 = s_2)^{\top}_{\mathrm{S}}; \\
\qquad \left(\mathtt{var}\ a_1; (a_1 = s_1)^{\top}_{\mathrm{S}}; (b_1 \vee b_2) * (b_1 \overrightarrow{;}_{\text{snc}} P_1 \parallel_{\hat{M}} b_2 \overrightarrow{;}_{\text{snc}} P_2); a_1 = f^{\perp}_{\mathrm{S}}\right)_{[a_2]}; (a_2 = s_2)^{\perp}_{\mathrm{S}}; \mathsf{end}\ a_1, a_2 \\
\sqsubseteq \{\text{Laws 3.3.90, 3.3.87, 3.3.92, 3.3.92, 3.3.75, 3.3.47 and 3.3.45}\} \\
\quad \mathtt{var}\ a_1, a_2; (a_1 = s_1 \wedge a_2 = s_2)^{\top}_{\mathrm{S}}; \\
\qquad \left((b_1 \vee b_2) * (b_1 \overrightarrow{;}_{\text{snc}} P_1 \parallel_{\hat{M}} b_2 \overrightarrow{;}_{\text{snc}} P_2)\right)_{[a_2]}; (a_1 = f \wedge a_2 = s_2)^{\perp}_{\mathrm{S}}; \mathsf{end}\ a_1, a_2 \\
\sqsubseteq \{\text{Laws 3.3.171, 3.3.91, 3.3.147 and 3.3.159}\} \\
\quad \mathtt{var}\ a_1, a_2; (a_1 = s_1 \wedge a_2 = s_2)^{\top}_{\mathrm{S}}; \\
\qquad ((b_1 \vee b_2) \wedge a_2 = s_2) * (b_1 \overrightarrow{;}_{\text{snc}} P_{1[a_2]} \parallel_{\hat{M}} b_2 \overrightarrow{;}_{\text{snc}} P_{2[a_2]}); (a_1 = f \wedge a_2 = s_2)^{\perp}_{\mathrm{S}}; \mathsf{end}\ a_1, a_2
\end{array}
$$

$$= \{\text{Definition 5.1.17}\}$$
$$a_1, a_2 : [a_1 = s_1 \wedge a_2 = s_2, \left(b_1 \underset{\text{snc}}{\rightarrow} P_{1[a_2]} \parallel_{\hat{M}} b_2 \underset{\text{snc}}{\rightarrow} P_{2[a_2]}\right), a_1 = f \wedge a_2 = s_2] \qquad \square$$

Note that the modification to the precedent normal form will not affect the behaviour of the normal form representing the parallel assignment. In fact, the latter will only activate its first step when both $a_1 = f$ and $a_2 = s_2$. The way we have constructed the normal form ensures that $a_1 = f$ will hold only after the normal form representing $x \overset{:=}{\underset{\text{HC}}{}} e_0$ has finished.

5.2 Parallel-by-merge elimination

Given Handel-C's shared-variable nature, when several normal forms are put together it is highly possible that more than one of them will be updating the same variable. Furthermore, due to the semantics of our guarded commands, all program variables are updated by each step, even when they are not activated[2]. This simultaneous attempt to update the program variables is one of the reasons that led us to introduce parallel by merge as the combinator for steps in our fist normal form. As mentioned before, the idea behind our parallel by merge operator is that each of the parallel processes has a private copy of the shared resources that it is allowed to modify and that all copies get synchronised at the end of each clock cycle.

Unfortunately, the much simpler parallelism provided by hardware devices cannot directly perform all these actions *natively* (i.e., the parallel behaviour the FPGA can produce is not expressive enough to directly execute processes in parallel-by-merge). This fact forced us to provide a way of simplifying the parallelism present in our first normal form into a form that can be directly mapped to the lower level parallelism available in hardware devices.

Our strategy to is to generate hardware mimicking the implementation of parallel-by-merge to compensate for the actions the FPGA cannot perform. Given the level of abstraction and complexity of the execution of $P \parallel_{\hat{M}} Q$, this process is only possible for simple cases of P and Q. Fortunately, steps in our first normal form can be expressed as just conditional, one-clock cycle assignments (see next section for details), allowing us to simplify parallel-by-merge with the strategy described above. A consequence of this final observation is that parallel-by-merge elimination can only be performed once the whole program has been reduced to first normal form, forcing us to introduce a second normal form in order to be able to serialise the compilation process.

The goal of this section is to describe our second normal form and the strategy that allows us to simplify the first normal form parallel behaviour.

5.2.1 Second normal form

In a general state machine, expressions are evaluated combinatorially and their results are then used to update the variables in the device's memory at the end of the clock cycle. Furthermore,

[2]Remember that guarded commands reduce to $\mathbb{I}$; sync when their condition does not hold. This means the step is trying to set all variables to be equal to themselves during that clock cycle.

dedicated hardware is used to compute the value of expressions and the results are then transmitted via wires to the flip-flops storing the program variables.

In our first normal form, all this behaviour is encoded within machine steps that are, in essence, guarded multiple assignments. In fact, each step calculates an update value for each of the shared variables at every clock cycle. The final value for a variable x at the end of each clock cycle is calculated by means of merging the values provided by all the steps modifying it. On the other hand, the value calculated by each step to update x is, most of the times, the previous value of the variable itself (this is the case when the guard is not satisfied and the whole step is reduced to $I\!I_1$).

The goal of this section is to reduce the normal form from the previous section into a new normal form comprising a **single step**. Let w be a list of special *wire* variables and v the list of program variables including the normal form control variables. The step for our second normal form is defined as follows:

Definition 5.2.1. *Second normal form step*

$$w \overset{:=}{_{snc}} f_w(v); (v \overset{:=}{_{snc}} f_v(w))_1$$

where:

- $f_w(v)$ *is a function updating the wires* w *based on the program and control variables in* v*.*
- $f_v(w)$ *is a function updating the program and control variables in* v *using the values carried within the wires in* w*.*

In turn, our second normal form also declares the wires that transmit values from the combinatorial logic calculating the value of expressions into the storage area where variables are stored. As with our first normal form, the execution of the second normal form is dependant on the variables and wires holding particular initial values. The assumption about the initial value for the wires is that they are equal to the logical value **false** and the buses have the value `ARB`. More formally, we define the second normal form for the compilation as follows.

Definition 5.2.2. *Second normal form*

$$a, w : [s, b * (P), f] =_{df} \texttt{var}\ a, w; (s)^{\top}_{S}; b * P; (f)^{\perp}_{S}; \texttt{end}\ a, w$$

where P is a second normal form step, as described by Definition 5.2.1.

The rest of this section is devoted to show how to reduce the steps from the first normal form into the single step required by our second normal form. The results in Section 5.2.2 reduce all steps into a single step similar to the one in Definition 5.2.1 but with local wires. Finally, Section 5.2.3, extends the scope of the wires and presents the final transformation to reduce to second normal form.

5.2.2 Simplifying guarded multiple assignments

A simple inspection of the theorems in the previous section is enough to show that all steps in our first normal form are guarded assignments of the form $b \overset{\rightarrow}{_{\text{snc}}} (v \overset{:=}{_{\text{snc}}} e)_1$ where v is a list of variables.

In this section we first show how to transform guarded simultaneous assignments (i.e., when v is a list of variables) into the parallel by merge composition of single-variable assignments. After that, we show how to reduce steps further to a point where they are in the form described in Definition 5.2.1 but with the scope of the relevant wires local to the step.

We first observe that a guarded multiple assignment can be described as the parallel by merge composition of guarded assignments to each of its individual variables. More formally:

Lemma 5.2.3. *Transforming guarded multiple assignments*

$$b \overrightarrow{}_{snc}(x, y \mathrel{:=}_{snc} e_1, e_2)_1 = b \overrightarrow{}_{snc}(x \mathrel{:=}_{snc} e_1)_1 \parallel_{\hat{M}} b \overrightarrow{}_{snc}(y \mathrel{:=}_{snc} e_2)_1$$

provided x and y are different variables

Proof.

Straightforward from Definition 4.2.3 and law 3.3.155. □

In this way we can express all steps in our first normal form as guarded, single-variable assignments. The next step is to push the guard inside the expression and transform it into a conditional assignment. The following lemma captures this notion more precisely.

Lemma 5.2.4. *Simplifying guarded, single variable, assignments*

$$b \overrightarrow{}_{snc}(x \mathrel{:=}_{snc} e)_1 = (x \mathrel{:=}_{snc} e \lhd b \rhd x)_1$$

Proof.

$$\begin{aligned}
& b \overrightarrow{}_{snc}(x \mathrel{:=}_{snc} e)_1 \\
&= \{\text{Definitions 3.3.149 and 3.2.7}\} \\
& (x \mathrel{:=}_{snc} e)_1 \lhd b \rhd (x \mathrel{:=}_{snc} x)_1 \\
&= \{\text{Law 3.3.28}\} \\
& (x \mathrel{:=}_{snc} e \lhd b \rhd x)_1
\end{aligned}$$

□

Having a way of simplifying the steps in the normal form into single assignments, we are in good position to separate the evaluation of expressions from the actual update of the program variables with those values. This is achieved by means of introducing additional wires and transforming every step into the combination of two parts: (a) a modified version of the original step where updates are stored in the recently introduced wires, in sequential composition with (b) a process copying the value of the wires into the globally accessible store. Lemma 5.2.5 achieves the partitioning described above.

Lemma 5.2.5. *Single step parallel transformation*

$$(x \mathrel{:=}_{snc} e_1)_1 \sqsubseteq \mathtt{var}\ i.x; (i.x \mathrel{:=}_{snc} e_1); (x \mathrel{:=}_{snc} i.x)_1; \mathtt{end}\ i.x$$

Proof.

$$
\begin{array}{ll}
& \mathtt{var}\ i.x; i.x \mathrel{:=_{snc}} e_1; (x \mathrel{:=_{snc}} i.x)_1; \mathsf{end}\ i.x \\
= & \{\text{Laws 3.3.16 and 3.3.72}\} \\
& \mathtt{var}\ i.x; (x \mathrel{:=_{snc}} e_1)_1; \mathsf{end}\ i.x \\
\sqsupseteq & \{\text{Laws 3.3.94 and 3.3.70}\} \\
& (x \mathrel{:=_{snc}} e_1)_1
\end{array}
$$

□

Finally, we can address the elimination of the complex parallel by merge we have used as the combinator for our steps. As shown in Section 2.3.8.1, the merge predicate $M(x, i.x, j.x, x.in')$ can be expressed as a deterministic selection function of the form $\mathtt{SELECT}\,(x, i.x, j.x)$. With this observation in mind, Lemma 5.2.6 shows how to reduce the parallel by merge execution of two steps in the form above back to the form established by Lemma 5.2.5.

Lemma 5.2.6.

$$
\left(\begin{array}{l}
(\mathtt{var}\ i.x \mathrel{:=_{snc}} e_1; (x \mathrel{:=_{snc}} i.x)_1; \mathsf{end}\ i.x) \parallel_{\hat{M}} \\
(\mathtt{var}\ j.x \mathrel{:=_{snc}} e_2; (x \mathrel{:=_{snc}} j.x)_1; \mathsf{end}\ j.x)
\end{array}\right) \sqsubseteq \left(\begin{array}{l}
\mathtt{var}\ i.x, j.x; \\
\quad (i.x, j.x \mathrel{:=_{snc}} e_1, e_2); (x \mathrel{:=_{snc}} \mathtt{SELECT}\,(x, i.x, j.x))_1; \\
\mathsf{end}\ i.x, j.x
\end{array}\right)
$$

Proof.

$$
\begin{array}{ll}
& (\mathtt{var}\ i.x \mathrel{:=_{snc}} e_1; (x \mathrel{:=_{snc}} i.x)_1; \mathsf{end}\ i.x) \parallel_{\hat{M}} (\mathtt{var}\ j.x \mathrel{:=_{snc}} e_2; (x \mathrel{:=_{snc}} j.x)_1; \mathsf{end}\ j.x) \\
\sqsubseteq & \{\text{Laws 3.3.148, 3.3.75 and 3.3.76}\} \\
& \mathtt{var}\ i.x, j.x; (i.x \mathrel{:=_{snc}} e_1; (x \mathrel{:=_{snc}} i.x)_1) \parallel_{\hat{M}} (j.x \mathrel{:=_{snc}} e_2; (x \mathrel{:=_{snc}} j.x)_1); \mathsf{end}\ i.x, j.x \\
\sqsubseteq & \{\text{Laws 3.3.16, 3.3.10, 3.3.142 and 3.3.146, theorem 2.3.122}\} \\
& \mathtt{var}\ i.x, j.x; (x \mathrel{:=_{snc}} \mathtt{SELECT}\,(x, e_1, e_2))_1); \mathsf{end}\ i.x, j.x \\
= & \{\text{Laws 3.3.16 and 3.3.72}\} \\
& \mathtt{var}\ i.x, j.x; (i.x, j.x \mathrel{:=_{snc}} e_1, e_2); (x \mathrel{:=_{snc}} \mathtt{SELECT}\,(x, i.x, j.x))_1; \mathsf{end}\ i.x, j.x
\end{array}
$$

□

5.2.3 Reaching second normal form

The previous two sections presented the different stages followed in the transformation from first to second normal form. In summary, the transformation is as follows:

1. The separation of multiple assignment within a step into guarded single-variable assignments executed in parallel-by-merge (Lemma 5.2.3).

2. The transformation of guarded assignments into conditional assignments (Lemma 5.2.4).

3. The introduction of wires and the separation of variable assignments into the combinatorial calculation of the value of expressions and the actual update of the program variables. These

wires used to transmit the values from the combinatorial logic into the memory-capable hardware storing the program variables (Lemma 5.2.5).

4. The elimination of parallel-by-merge as the combinator of steps (Lemma 5.2.6).

The only missing aspect in the reduction to second normal form is the promotion of the declaration of the wires to outside the execution loop. As wires are special cases of variables, their scope can be expanded by means of Law 3.3.105, leading to refinement. The complete sequence of transformations is summarised by the following theorem, showing how to transform programs from first to second normal form.

Theorem 5.2.7. *First- to second-normal-form transformation*

$$a : [s, (b \overrightarrow{;}_{\mathsf{snc}} P), f] \sqsubseteq a, w : [s, b * (w \mathrel{:=}_{\mathsf{snc}} f_w(v); (v \mathrel{:=}_{\mathsf{snc}} f_v(w))_1), f]$$

where $P = b_1 \overrightarrow{;}_{\mathsf{snc}} (v_1 \mathrel{:=}_{\mathsf{snc}} g_1)_1 \parallel_{\hat{M}} \ldots \parallel_{\hat{M}} b_n \overrightarrow{;}_{\mathsf{snc}} (v_n \mathrel{:=}_{\mathsf{snc}} g_n)_1$

Proof.

$$\begin{aligned}
& a : [s, (b \overrightarrow{;}_{\mathsf{snc}} P), f] \\
= & \ \{\text{Definitions 5.1.17 and 5.1.8}\} \\
& \mathtt{var}\ a; s_{\mathsf{S}}^{\top}; b * b \overrightarrow{;}_{\mathsf{snc}} P; f_{\mathsf{S}}^{\bot}; \mathsf{end}\ a \\
= & \ \{\text{Lemmas 5.2.3 and 5.2.4 (where } e_1 = v_1 \lhd b_1 \rhd g_1, \ldots, e_n = v_n \lhd b_n \rhd g_n)\} \\
& \mathtt{var}\ a; s_{\mathsf{S}}^{\top}; b * \left((v_1 \mathrel{:=}_{\mathsf{snc}} e_1)_1 \parallel_{\hat{M}} \ldots \parallel_{\hat{M}} (v_n \mathrel{:=}_{\mathsf{snc}} e_n)_1\right); f_{\mathsf{S}}^{\bot}; \mathsf{end}\ a \\
= & \ \{\text{Lemmas 5.2.5 and 5.2.6 } (w = \langle w_1, \ldots, w_n \rangle, v = \langle v_1, \ldots, v_n \rangle \text{ and } f_w(v) = \langle e_1, \ldots, e_n \rangle)\} \\
& \mathtt{var}\ a; s_{\mathsf{S}}^{\top}; b * (\mathtt{var}\ w; w \mathrel{:=}_{\mathsf{snc}} f_w(v); (v \mathrel{:=}_{\mathsf{snc}} f_v(w))_1; \mathsf{end}\ w); f_{\mathsf{S}}^{\bot}; \mathsf{end}\ a \\
\sqsubseteq & \ \{\text{Laws 3.3.105, 3.3.92, 3.3.75, 3.3.93 and 3.3.76}\} \\
& \mathtt{var}\ a, w; s_{\mathsf{S}}^{\top}; b * (w \mathrel{:=}_{\mathsf{snc}} f_w(v); (v \mathrel{:=}_{\mathsf{snc}} f_v(w))_1); f_{\mathsf{S}}^{\bot}; \mathsf{end}\ a, w \\
= & \ \{\text{Definition 5.2.2}\} \\
& a, w : [s, b * (w \mathrel{:=}_{\mathsf{snc}} f_w(v); (v \mathrel{:=}_{\mathsf{snc}} f_v(w))_1), f]
\end{aligned}$$

□

5.3 The compilation strategy in action

The goal of this section is to illustrate how the compilation process described in the previous section works with some examples.

Example 5.3.1. *Assignments in parallel*

Consider now the slightly more complicated pair of assignments that are executed in parallel:

$$(x \mathrel{:=}_{\mathsf{HC}} e_1) \parallel_{\mathsf{HC}} (y \mathrel{:=}_{\mathsf{HC}} e_2)$$

The above program produces the parallel update of x and y (assumed to be distinct variables), with values e_1 and e_2 respectively) and terminates after one clock cycle. In our first normal form, this is achieved by combining two assignment normal forms using theorem 5.1.30.

$$a_0, a_1 : \left[\begin{array}{l} (a_0 = s_0) \wedge (a_1 = s_1), \\ \quad ((a_0 = s_0) \wedge (a_1 = s_1)) \overrightarrow{\ }_{\mathsf{snc}} (x, y, a_0, a_1 \mathrel{:=}_{\mathsf{snc}} e_1, e_2, f_0, f_1)_1), \\ (a_0 = f_0) \wedge (a_1 = f_1) \end{array}\right]$$

This first normal form can be further simplified and transformed into second normal form by means of introducing wires and separating the calculation of e_1, e_2 and the next control state from the actual updates of x, y, a_0 and a_1:

$$\begin{array}{l} \mathtt{var}\ a_0, a_1, w_{a_0}, w_{a_1}, w_x, w_y; \\ \quad ((a_0 = s_0) \wedge (a_1 = s_1))^{\top}_{\mathsf{S}}; \\ \quad (a_0 = s_0) \wedge (a_1 = s_1) * \left(\begin{array}{l} \left(\begin{array}{l} w_{a_0} \mathrel{:=}_{\mathsf{snc}} f_0 \lhd (a_0 = s_0) \wedge (a_1 = s_1) \rhd a_0 \parallel \\ w_{a_1} \mathrel{:=}_{\mathsf{snc}} f_1 \lhd (a_0 = s_0) \wedge (a_1 = s_1) \rhd a_1 \parallel \\ w_x \mathrel{:=}_{\mathsf{snc}} e_1 \lhd (a_0 = s_0) \wedge (a_1 = s_1) \rhd x \parallel \\ w_y \mathrel{:=}_{\mathsf{snc}} e_2 \lhd (a_0 = s_0) \wedge (a_1 = s_1) \rhd y \end{array}\right); \\ (a_0, a_1, x, y \mathrel{:=}_{\mathsf{snc}} w_{a_0}, w_{a_1}, w_x, w_y)_1 \end{array}\right); \\ \quad ((a_0 = f_0) \wedge (a_1 = f_1))^{\bot}_{\mathsf{S}}; \\ \mathtt{end}\ a_0, a_1, w_{a_0}, w_{a_1}, w_x, w_y \end{array}$$

Example 5.3.2. *Sequential composition of parallel and non-parallel normal forms*

We now address the compilation of a parallel assignment followed by a simple assignment:

$$(x \mathrel{:=}_{\mathsf{HC}} e_1 \parallel_{\mathsf{HC}} y \mathrel{:=}_{\mathsf{HC}} e_2) \fatsemi x \mathrel{:=}_{\mathsf{HC}} x + y$$

The above program expands our previous example to use the values assigned to x and y to update x itself. The program terminates after two clock cycles with $y = e_2$ and $x = e_1 + e_2$. The equivalent program in our first normal form is as follows:

$$a_0, a_1 : \left[\begin{array}{l} (a_0 = s_0) \wedge (a_1 = s_1), \\ \quad \left(\begin{array}{l} \quad (a_0 = s_0) \wedge (a_1 = s_1) \overrightarrow{\ }_{\mathsf{snc}} (x, y, a_0, a_1 \mathrel{:=}_{\mathsf{snc}} e_1, e_2, f_0, f_1)_1 \\ \parallel_{\hat{M}} (a_0 = f_0) \wedge (a_1 = f_1) \overrightarrow{\ }_{\mathsf{snc}} (x, a_0 \mathrel{:=}_{\mathsf{snc}} x + y, f_2)_1 \end{array}\right), \\ (a_0 = f_2) \wedge (a_1 = f_1) \end{array}\right]$$

Note that the control variables for $x \mathrel{:=}_{\mathsf{HC}} x + y$ have been extended to incorporate a_1 (our normal form compilation strategy initially generated a normal form for this construct that had a_0 as its

only control variable). It is also important to notice that the last step keeps y and a_1 constant (i.e., $y, a_1 \overset{:=}{_{\text{snc}}} y, a_1$) as the the assignment operator performs this additional assignments implicitly. We can now transform the state machine above into second normal form to eliminate parallel by merge:

$$
\begin{array}{l}
\texttt{var}\ a_0, a_1, 0.w_{a_0}, 1.w_{a_0}, w_{a_1}, 0.w_x, 1.w_x, w_y; \\
\quad ((a_0 = s_0) \wedge (a_1 = s_1))^{\top}_{\mathsf{S}}; \\
\quad \left(\begin{array}{c} (a_0 = s_0) \wedge (a_1 = s_1) \\ \vee \\ (a_0 = f_0) \wedge (a_1 = f_1) \end{array} \right) * \left(\begin{array}{l} \left(\begin{array}{l} 0.w_{a_0} \overset{:=}{_{\text{snc}}} f_0 \lhd (a_0 = s_0) \wedge (a_1 = s_1) \rhd a_0 \parallel \\ 1.w_{a_0} \overset{:=}{_{\text{snc}}} f_2 \lhd (a_0 = f_0) \wedge (a_1 = f_1) \rhd a_0 \parallel \\ w_{a_1} \overset{:=}{_{\text{snc}}} f_1 \lhd (a_0 = s_0) \wedge (a_1 = s_1) \rhd a_1 \parallel \\ 0.w_x \overset{:=}{_{\text{snc}}} e_1 \lhd (a_0 = s_0) \wedge (a_1 = s_1) \rhd x \parallel \\ 1.w_x \overset{:=}{_{\text{snc}}} x + y \lhd (a_0 = f_0) \wedge (a_1 = f_1) \rhd x \parallel \\ w_y \overset{:=}{_{\text{snc}}} e_2 \lhd (a_0 = s_0) \wedge (a_1 = s_1) \rhd y \end{array} \right); \\ \left(\begin{array}{l} a_0 \overset{:=}{_{\text{snc}}} \texttt{SELECT}\,(a_0, 0.w_{a_0}, 1.w_{a_0}) \parallel \\ a_1 \overset{:=}{_{\text{snc}}} w_{a_1} \parallel \\ x \overset{:=}{_{\text{snc}}} \texttt{SELECT}\,(a_0, 0.w_x, 1.w_x) \parallel \\ y \overset{:=}{_{\text{snc}}} w_y \end{array} \right)_1 \end{array} \right); \\
\quad ((a_0 = f_2) \wedge (a_1 = f_1))^{\bot}_{\mathsf{S}}; \\
\texttt{end}\ a_0, a_1, 0.w_{a_0}, 1.w_{a_0}, w_{a_1}, 0.w_x, 1.w_x, w_y
\end{array}
$$

Example 5.3.3. *Sequential composition within parallel branches*

Our final example illustrates the real parallelism that can be achieved in our normal form. The program is a sequence of assignments in parallel with a single assignment:

$$(x \overset{:=}{_{\mathsf{HC}}} e_1 \fatsemi x \overset{:=}{_{\mathsf{HC}}} x + y) \overset{\parallel}{_{\mathsf{HC}}} y \overset{:=}{_{\mathsf{HC}}} e_2$$

The above program illustrates two notions: (a) the fact that shared variables can be used to transmit values across parallel branches; and (b) our compilation strategy for parallel composition allows the different parallel branches to execute independently from each other. Expanding the semantics of the above program and applying theorems 5.1.20, 5.1.23 and 5.1.30 we can reduce the above program into the following first normal form:

$$
a_0, a_1 : \left[\begin{array}{l} (a_0 = s_0) \wedge (a_1 = s_1), \\ \quad \left(\begin{array}{l} \quad (a_0 = s_0) \wedge (a_1 = s_1) \overset{\rightarrow}{_{\text{snc}}} (x, y, a_0, a_1 \overset{:=}{_{\text{snc}}} e_1, e_2, s_0 + 1, s_0 + 1)_1 \\ \parallel_{\hat{M}} (a_0 = s_0 + 1) \overset{\rightarrow}{_{\text{snc}}} (x, a_0 \overset{:=}{_{\text{snc}}} x + y, s_0 + 2)_1 \end{array} \right), \\ (a_0 = s_0 + 2) \wedge (a_1 = s_0 + 1) \end{array} \right]
$$

The second step in the normal form above models the assignment $x \overset{:=}{_{\mathsf{HC}}} x + y$ in the original program. It is important to notice that the condition guarding this step only refers to the value of a_0, allowing any possible step depending on a_1 to be executed in parallel with it. The fact that the control value $s_0 + 1$ is unique and will only be assigned to a_0 after the previous step has been executed ensures a correct ordering of execution among the different steps in the normal form.

Again, we can eliminate the parallel by merge from the above state machine by means of reducing it to second normal form:

$$
\begin{array}{l}
\mathtt{var}\ a_0, a_1, 0.w_{a_0}, 1.w_{a_0}, w_{a_1}, 0.w_x, 1.w_x, w_y; \\
\quad ((a_0 = s_0) \wedge (a_1 = s_1))^{\top}_{\mathsf{S}}; \\
\quad \left(\begin{array}{c} (a_0 = s_0) \wedge (a_1 = s_1) \\ \vee \\ a_0 = s_0 + 1 \end{array} \right)^{*} \left(\begin{array}{l}
\left(\begin{array}{l}
0.w_{a_0} \overset{:=}{_{\mathsf{snc}}} s_0 + 1 \lhd (a_0 = s_0) \wedge (a_1 = s_1) \rhd a_0 \parallel \\
1.w_{a_0} \overset{:=}{_{\mathsf{snc}}} s_0 + 2 \lhd a_0 = s_0 + 1 \rhd a_0 \parallel \\
w_{a_1} \overset{:=}{_{\mathsf{snc}}} s_0 + 1 \lhd (a_0 = s_0) \wedge (a_1 = s_1) \rhd a_1 \parallel \\
0.w_x \overset{:=}{_{\mathsf{snc}}} e_1 \lhd (a_0 = s_0) \wedge (a_1 = s_1) \rhd x \parallel \\
1.w_x \overset{:=}{_{\mathsf{snc}}} x + y \lhd a_0 = s_0 + 1 \rhd x \parallel \\
w_y \overset{:=}{_{\mathsf{snc}}} e_2 \lhd (a_0 = s_0) \wedge (a_1 = s_1) \rhd y
\end{array} \right); \\
\left(\begin{array}{l}
a_0 \overset{:=}{_{\mathsf{snc}}} \mathtt{SELECT}\,(a_0, 0.w_{a_0}, 1.w_{a_0}) \parallel \\
a_1 \overset{:=}{_{\mathsf{snc}}} w_{a_1} \parallel \\
x \overset{:=}{_{\mathsf{snc}}} \mathtt{SELECT}\,(a_0, 0.w_x, 1.w_x) \parallel \\
y \overset{:=}{_{\mathsf{snc}}} w_y
\end{array} \right)_1
\end{array} \right); \\
\quad ((a_0 = s_0 + 2) \wedge (a_1 = s_0 + 1))^{\perp}_{\mathsf{S}}; \\
\mathtt{end}\ a_0, a_1, 0.w_{a_0}, 1.w_{a_0}, w_{a_1}, 0.w_x, 1.w_x, w_y
\end{array}
$$

5.4 Mapping the normal form into hardware

The goal of this section is to define a way of mapping our second normal form into a hardware description suitable for programming a Field Programmable Gate Array (FPGA). An FPGA is an integrated circuit designed to be configured by the customer or designer after manufacturing. FPGAs contain programmable logic components called *logic blocks*, and a hierarchy of reconfigurable interconnects that allow the blocks to be connected to each other. Logic blocks can be configured to perform any task that could be performed in hardware, from complex combinational functions to simple logic gates. In most FPGAs, the logic blocks also include memory elements, which may be simple flip-flops or more complete blocks of memory.

The simplest way to achieve this goal is to provide a way of expressing our final normal form as a low-level hardware description in VHDL, Verilog or any other architectural HDL. This approach has been successfully applied to generate hardware implementations of high level functions in ML or variants of Haskell [Iyoda 2007; Claessen and Pace 2002]. In the case of our compiler, the first and second normal forms resemble, in structure, the standard implementation of state machines

in VHDL [Abdel-hamid et al. 2004]. The problem of this approach is the already mentioned lack of suitable denotational semantics for VHDL (see Section 1.2.4) that could be integrated with our approach. Furthermore, the synthesis of this kind of VHDL architectural descriptions would have to be verified if we are to maintain our goal of a correct compiler for Handel-C.

Another common strategy to achieve this goal would be to select an FPGA provider (e.g., Xilinx, Altera or Actel) and show how to map the final normal form into the vendor's proprietary net-list language. This approach is based on the assumption that hardware components that can be instantiated in the net-list are correctly implemented and simple enough for their semantics to be straightforward. Even though this strategy has proven itself efficient and successful [Iyoda and He 2001*b*;*a*; Susanto and Melham 2001] it binds the compiler to a specific vendor and limits the number of tools that can be used afterwards when synthesising hardware (mainly optimisation and place and route algorithms).

Based on the limitations of the two approaches described before, we have decided to provide a way of mapping our final normal form into generic, basic hardware components (the same kind of components available in most proprietary net-list languages). In this way, we take advantage of the successful experiences of previous hardware-generating tools without binding our compiler to a specific vendor.

5.4.1 Generating hardware for the final normal form's step

As described in Section 5.2, our second normal form is expressed in terms of the iteration of a single step in the following format:

$$(w_1 \overset{:=}{_{\mathrm{snc}}} (e_1 \lhd c_1 \rhd x) \parallel \cdots \parallel w_n \overset{:=}{_{\mathrm{snc}}} (e_n \lhd c_n \rhd x)) \,;\, (x \overset{:=}{_{\mathrm{snc}}} \mathtt{SELECT}\,(x, \mathtt{SELECT}\,(x, \ldots, w_{n-1}), w_n))_1$$

The hardware-mapping of this kind of step can be split into two main stages. Firstly, we need to associate hardware to the combinatorial aspects of the expression above (i.e., the sub-expression on the left of the sequential composition within the step). This particular part of the step can be precisely described as the parallel composition of independent updates to wires of the form:

$$w \overset{:=}{_{\mathrm{snc}}} (e \lhd c \rhd v)$$

where v is one of the program/control variables and c is a control-based condition. The generation of hardware for this aspect of the combinatorial part of a clock cycle can be described as follows:

1. Generate combinatorial hardware to calculate the value expression e.
2. Generate combinatorial logic to evaluate condition c.
3. Use the allocated wires w, w_v, w_e and w_c and associate them with w, the value of register v and the evaluations of e and c respectively (see Section 5.4.2.3 for details on how the wires are allocated).
4. Instantiate a two-way multiplexor to select from the value calculated for e and the value of v. The wires introduced in step (3) are used to transfer the values from the corresponding

source hardware into the multiplexer. The multiplexer is controlled by the result of the evaluation of c (transferred to the multiplexer by means of wire w_c). The outcome of the multiplexer is to be transported by wire w.

The implementation of expressions in combinatorial logic is achieved by means of combining functional units (i.e., adders, subtracters) and logical gates. The whole process is illustrated in more detail in Figure 5.1.

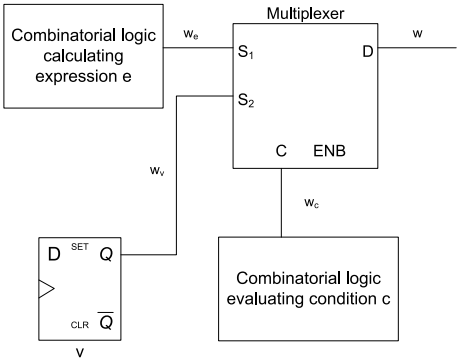

Figure 5.1: Hardware mapping of combinatorial part of a step

The remaining part of the normal form's step deals with the updating of the program and control variables stored in memory-capable devices within the FPGA. From the generic step described above, this part can be seen as the parallel combination of one-clock cycle assignments of the form:

$$(v \underset{\text{snc}}{:=} \texttt{SELECT}\,(v, w_1, w_2))_1$$

Taking into account that the allocation of wires w_v, w_1 and w_2 has already been performed by the combinatorial aspects of the machine, the process of generating hardware for this part of the step can be outlined as follows:

1. Use allocated wire $w_{\texttt{SELECT}(v)}$ to transmit the merged value that will be used to update v.
2. Generate a SELECT block (see below). The inputs for this instance would be w_v (reference value), and w_1 and w_2 (wires transmitting the results of two processes trying to update v in the program). The output of the selector is carried into the memory cell storing v by means of the $w_{\texttt{SELECT}(v)}$ wire introduced in step (1).

The selection function was defined as the predicate (see Definition 2.3.121):

$$\texttt{SELECT}\,(v, 0.v, 1.v) \;=_{df}\; (v \lhd v = 1.v \rhd 1.v) \lhd v = 0.v \rhd 0.v$$

It is easy to see that when $1.v = v$, the sub expression $(v \lhd v = 1.v \rhd 1.v)$ is always equal to $1.v$, allowing us to simplify the above definition into the form:

$$\texttt{SELECT}\,(v, 0.v, 1.v) \;=_{df}\; 1.v \lhd v = 0.v \rhd 0.v$$

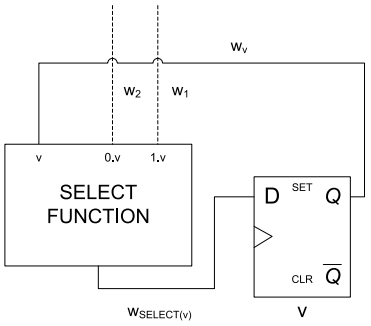

Figure 5.2: Generated hardware for the sequential part of the step

From this simplified definition, the SELECT function can be implemented in hardware as shown in Figure 5.3 (assuming the multiplexer selects s_1 when its control has the logical value **true**).

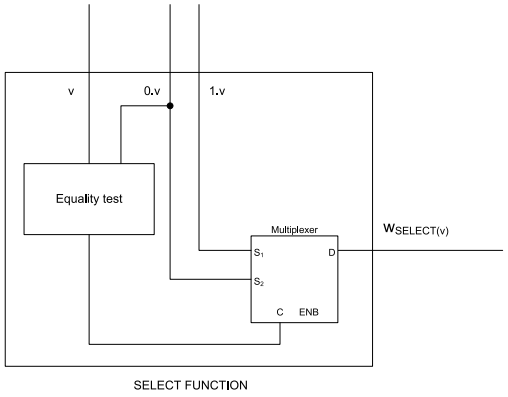

Figure 5.3: Implementation of the SELECT function

Finally, nested selection functions of the form

$$\texttt{SELECT}\,(v, \texttt{SELECT}\,(v, w_1, w_2), w_3)$$

are implemented in hardware by means of chaining the binary implementation of the selection function described above. The additional wire $w_{\texttt{SELECT}(w_1,w_2)}$ is used to transfer the intermediate result of the inner application of the selection function into the containing one. The same approach can be generalised for selection functions of deeper nesting.

All the wires used for the transferral of values between combinatorial fragments will be allocated when treating the remaining aspects of the normal form (see Section 5.4.2.3).

5.4.2 Encoding the rest of the second normal form

Having a way of encoding the step describing the execution of our machine, we now concentrate on the remaining aspects of the second normal form we have not yet mapped into hardware. Recasting from our definition for the second normal form

$$a, w : [s, c_a * (P), f] =_{df} \texttt{var}\ a, w; (s)^{\top}_{\mathsf{S}}; c_a * P; (f)^{\perp}_{\mathsf{S}}; \texttt{end}\ a, w$$

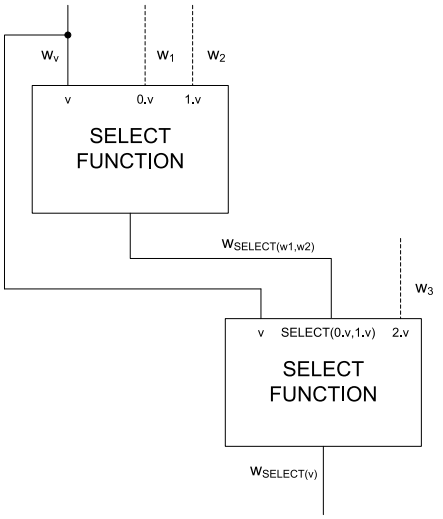

Figure 5.4: Multiple-argument SELECT function

where P is a second normal form step, as described by Definition 5.2.1, we need to address the hardware generation of the following aspects:

1. The execution loop
2. The assumptions and assertions controlling the execution loop's initial and final state.
3. The declaration of basic wires and storage for program variables.

The following subsections address the generation of hardware for the different aspects of the normal form mentioned above.

5.4.2.1 The execution loop

A state machine like the one described by our normal forms (i.e., a machine with an internal storage and where the next state depends on the stored values and, possibly, on some inputs) can be seen as having two disjoint sets of behaviours: (a) an *active phase* where the control state matches one or more of the conditions activating part of the combinatorial logic (effectively deriving in the calculation of a new state and changing the machine's control flow); and (b) a *passive phase* where there is no combinatorial behaviour associated to the current state and the machine remains in the same control state. In this context, a key observation is that once (b) is reached, the machine will remain in the same control state attempting to make progress at every clock cycle until the machine breaks or it is powered off.

It is easy to see that the execution loop in our normal form describes the behaviour of the machine when it is in the state (a) above, where the (b) part is implicit in the normal form. Note that, provided we denote the control conditions associated with (a) and (b) with c_a and c_b respectively, then we have that $\neg(c_a \wedge c_b)$ (as (a) and (b) are disjoint). In this way, the normal form above can be seen as the equation

$$a, w : [s, c_a * (P), f] =_{df} \texttt{var}\ a, w; (s)^{\top}_{\mathrm{S}}; c_a * P; c_b * \mathbb{I}_1; (f)^{\bot}_{\mathrm{S}}; \texttt{end}\ a, w$$

and thanks to Laws 5.1.14 and 3.3.145 it can be further reduced to

$$a, w : [s, c_a * (P), f] =_{df} \texttt{var}\ a, w; (s)^{\top}_{\mathsf{S}}; (c_a \vee c_b) * P; (f)^{\perp}_{\mathsf{S}}; \texttt{end}\ a, w$$

The key consequence of the result above is that the normal form precisely captures the behaviour of a state machine when implemented in hardware. Furthermore, no hardware needs to be generated to explicitly capture the iterated step of the normal form, as this behaviour will be implicitly accomplished by the way in which the hardware is generated.

To illustrate this idea, let us consider the simple example we introduced in Chapter 1:

$$x \overset{:=}{_{\mathsf{HC}}} e$$

that generates the following expanded final normal form

$$\begin{array}{l}
\texttt{var}\ w_c, w_a, w_x, a; \\
\quad (a = s)^{\top}_{\mathsf{S}}; \\
\quad (a = s) * \left(\begin{array}{l} w_c \overset{:=}{_{\mathsf{snc}}} a = s; \\ w_x \overset{:=}{_{\mathsf{snc}}} (e \lhd w_c \rhd x) \parallel w_a \overset{:=}{_{\mathsf{snc}}} (f \lhd w_c \rhd a); \\ (x, a \overset{:=}{_{\mathsf{snc}}} w_x, w_a)_1 \end{array} \right); \\
\quad (a = f)^{\perp}_{\mathsf{S}}; \\
\texttt{end}\ w_c, w_a, w_x, a
\end{array}$$

The execution of the above normal form spends the first clock cycle setting x to the value calculated from evaluating e and a to f. In the following clock cycle (and every other clock cycle after that one), the machine does nothing but keeping x and a constant. This same behaviour is achieved by the hardware generated for the normal form, as shown in Figure 5.5.

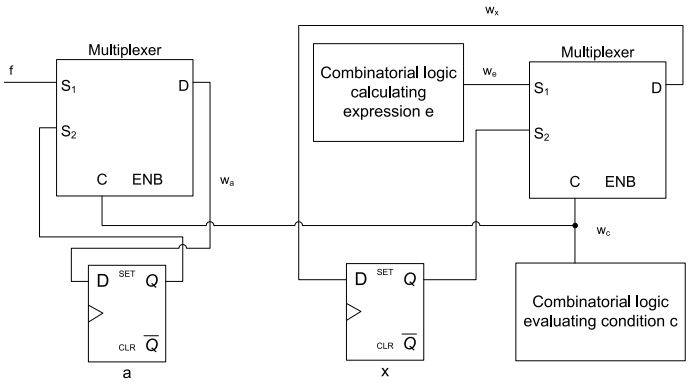

Figure 5.5: Compiled iterated step for $x \overset{:=}{_{\mathsf{HC}}} e$

5.4.2.2 Assumptions and assertions

The assumption at the beginning of the normal form ensures the machine starts by executing its first step(s). The semantics of the assumption construct ensures that if the machine is not started in that particular control state, the behaviour of the whole normal form will be miraculous. As it is not possible to implement a *miracle*, we need a way of transforming the assumption of initial values into a different construct we can implement. Fortunately, Law 3.3.74 establishes the very useful equivalence

$$\texttt{var}\ x; (x = e)^{\top}_{\mathrm{S}} = \texttt{var}\ x; x \overset{:=}{_{\mathrm{snc}}} e$$

This law is telling us that assuming a new variable x will have a special initial value e is equivalent to actually initialising x to e. This is exactly what we are looking for as we can now eliminate the miraculous behaviour we were not able to represent at the hardware level.

On the other hand, the assertion at the end of the execution loop was only introduced in the normal form for the sake of reasoning and combination of normal forms. This information is no longer necessary when mapping the normal form into hardware so it can be safely abstracted out. The desired effect can be achieved by refining our normal form by means of Law 3.3.40:

$$(x = e)^{\top}_{\mathrm{S}} \sqsubseteq \mathrm{II}$$

and the fact that II is the left unit for sequential composition. The two observations above allow us to refine our second normal form

$$a, w : [a = s, b * (P), a = f] =_{df} \texttt{var}\ a, w; (a = s)^{\top}_{\mathrm{S}}; b * P; (a = f)^{\bot}_{\mathrm{S}}; \texttt{end}\ a, w$$

into the simpler form:

$$\texttt{var}\ a, w; (a \overset{:=}{_{\mathrm{snc}}} s); b * P; \texttt{end}\ a, w$$

The hardware encoding of the initial assumption and final assertion in this new version of the normal form only requires us to set the initial values of the registers storing the control variables to the values required to trigger its first step.

5.4.2.3 Wire and storage allocation

The allocation of wires and storage for program and control variables follows from the definition of our final normal form as the list of variables and wires to be allocated is explicitly declared. There are, however, two issues that deserve clarification when allocating hardware for these aspects of our final normal form:

1. *Wire-completeness*. In the generation of the hardware associated with the combinatorial and sequential part of the step we have referred to wires that have not been explicitly described in the transformation from first to second normal form (e.g., the wire transferring the result

of the evaluation of the control-condition for each step). These 'missing' wires have only been omitted from the normal form in order to keep it in a format that is easier for the reader to understand. For example, it is easy to see that the combinatorial part of a step of the form:

$$w \overset{:=}{}_{\text{snc}} (e \lhd a = 0 \rhd x)$$

can be refined, by means of Lemma 5.2.5 into the more explicit form

$$\texttt{var}\ w_a; w_a \overset{:=}{}_{\text{snc}} a = 0; w \overset{:=}{}_{\text{snc}} (e \lhd w_a \rhd x); \texttt{end}\ w_a$$

allowing us to introduce the omitted wires and to keep the format required in the steps for the second normal form.

2. *Memory allocation*. Modern FPGA devices provide three ways of storing information: (a) synthesised memory blocks in specifically designed *slices* within the FPGA (i.e., implement a memory-capable register using the FPGA's logic); (b) allocate memory in one of the RAM blocks connected to the FPGA; or (c) distributed RAM, built by means of LookUp Table memory units within the FPGA. Even though the ideal solution would be to save as much programmable space as possible from the FPGA by using the RAM blocks for program and control variables, there is a limit in the number of parallel accesses that can be performed in RAM blocks. This restriction makes strategies (b) and (c) unfeasible in our context as an arbitrary number of variables need to be read and updated at each clock cycle.

5.5 Putting it all together: a sequence of assignments into hardware

In this section we illustrate the hardware-mapping strategy described above with a simple example. The code we have selected to compile into hardware is the following sequence of assignments:

$$x \overset{:=}{}_{\text{snc}} e \fatsemi y \overset{:=}{}_{\text{snc}} x + 1$$

The above program can be reduced to first normal form using theorems 5.1.20 and 5.1.23:

$$a : [a = s, \begin{pmatrix} (a = s) \overset{\rightarrow}{}_{\text{snc}} (a, x \overset{:=}{}_{\text{snc}} s + 1, e)_1 \\ \|_{\hat{M}}\ (a = s + 1) \overset{\rightarrow}{}_{\text{snc}} (a, y \overset{:=}{}_{\text{snc}} s + 2)_1 \end{pmatrix}, (a = s + 2)]$$

Following theorem 5.2.7 we can simplify the parallelism in the normal form:

$$
\begin{array}{l}
\texttt{var}\ a, w_{a1}, w_{a2}, w_x, w_y; \\
\quad (a = s)^{\top}_{\mathsf{S}} \\
\quad \left(\begin{array}{c} a = s \\ \vee \\ a = s + 1 \end{array} \right) * \left(\begin{array}{l} \left(\begin{array}{l} w_{a1} \overset{:=}{_{\mathsf{snc}}} s_0 + 1 \lhd a = s \rhd a \parallel \\ w_{a2} \overset{:=}{_{\mathsf{snc}}} s_0 + 2 \lhd (a = s + 1) \rhd a \parallel \\ w_x \overset{:=}{_{\mathsf{snc}}} e \lhd a = s \rhd x \parallel \\ w_y \overset{:=}{_{\mathsf{snc}}} x + 1 \lhd (a = s + 1) \rhd y \end{array} \right); \\ \left((a \overset{:=}{_{\mathsf{snc}}} \mathsf{SELECT}\,(a_0, w_{a1}, w_{a2})) \parallel (x \overset{:=}{_{\mathsf{snc}}} w_x) \parallel (y \overset{:=}{_{\mathsf{snc}}} w_y) \right)_1 \end{array} \right); \\
\quad (a = s + 2)^{\bot}_{\mathsf{S}} \\
\texttt{end}\ a, w_{a1}, w_{a2}, w_x, w_y
\end{array}
$$

Our compilation theorems together with the simplification of the initial assumption and final assertion, reduce the above program into the following form:

$$
\begin{array}{l}
\texttt{var}\ a_w, x_w, y_w, w_{c1}, w_{c2}, w_e, w_{x+1}, w_{s+1}, w_{s+2}, w_{\mathsf{SELECT}(a)}, w_x, w_y, w_{a1}, w_{a2}, a \overset{:=}{_{\mathsf{snc}}} s; \\
\quad (a = s) * \left(\begin{array}{l} \left(\begin{array}{l} a_w \overset{:=}{_{\mathsf{snc}}} a \parallel x_w \overset{:=}{_{\mathsf{snc}}} x \parallel y_w \overset{:=}{_{\mathsf{snc}}} y \parallel \\ w_{c1} \overset{:=}{_{\mathsf{snc}}} a_w = s \parallel w_{c2} \overset{:=}{_{\mathsf{snc}}} a_w = s + 1 \parallel \\ w_e \overset{:=}{_{\mathsf{snc}}} e \parallel w_{x+1} \overset{:=}{_{\mathsf{snc}}} x_w + 1 \parallel w_{s+2} \overset{:=}{_{\mathsf{snc}}} s + 2 \end{array} \right); \\ \left(\begin{array}{l} w_x \overset{:=}{_{\mathsf{snc}}} (w_e \lhd w_{c1} \rhd x_w) \parallel w_y \overset{:=}{_{\mathsf{snc}}} (w_{x+1} \lhd w_{c2} \rhd y_w) \parallel \\ w_{a1} \overset{:=}{_{\mathsf{snc}}} (w_{s+1} \lhd w_{c1} \rhd a_w) \parallel w_{a2} \overset{:=}{_{\mathsf{snc}}} (w_{s+2} \lhd w_{c2} \rhd a_w) \end{array} \right); \\ w_{\mathsf{SELECT}(a)} \overset{:=}{_{\mathsf{snc}}} \mathsf{SELECT}\,(a_w, w_{a1}, w_{a2}); \\ (x, y, a \overset{:=}{_{\mathsf{snc}}} w_x, w_y, w_{\mathsf{SELECT}(a)})_1 \end{array} \right); \\
\texttt{end}\ a_w, x_w, y_w, w_{c1}, w_{c2}, w_e, w_{x+1}, w_{s+1}, w_{s+2}, w_{\mathsf{SELECT}(a)}, w_x, w_y, w_{a1}, w_{a2}, a
\end{array}
$$

Using the approach described in the previous section, the above modified second normal form can be mapped into the hardware as described by Figure 5.6.

5.6 Chapter summary

In this chapter we have covered the following topics:

- **First normal form**. Our first normal form was defined to resemble a state-machine encoding of the source program. The first normal form is comprised by an assumption about its initial step, an iteration over a set of steps that carry out the computations in the program, and an assertion about the control state the machine should be in when it finishes executing. Parallelism is encoded in the normal form by means of allowing more than one execution step to be active at any given clock cycle.

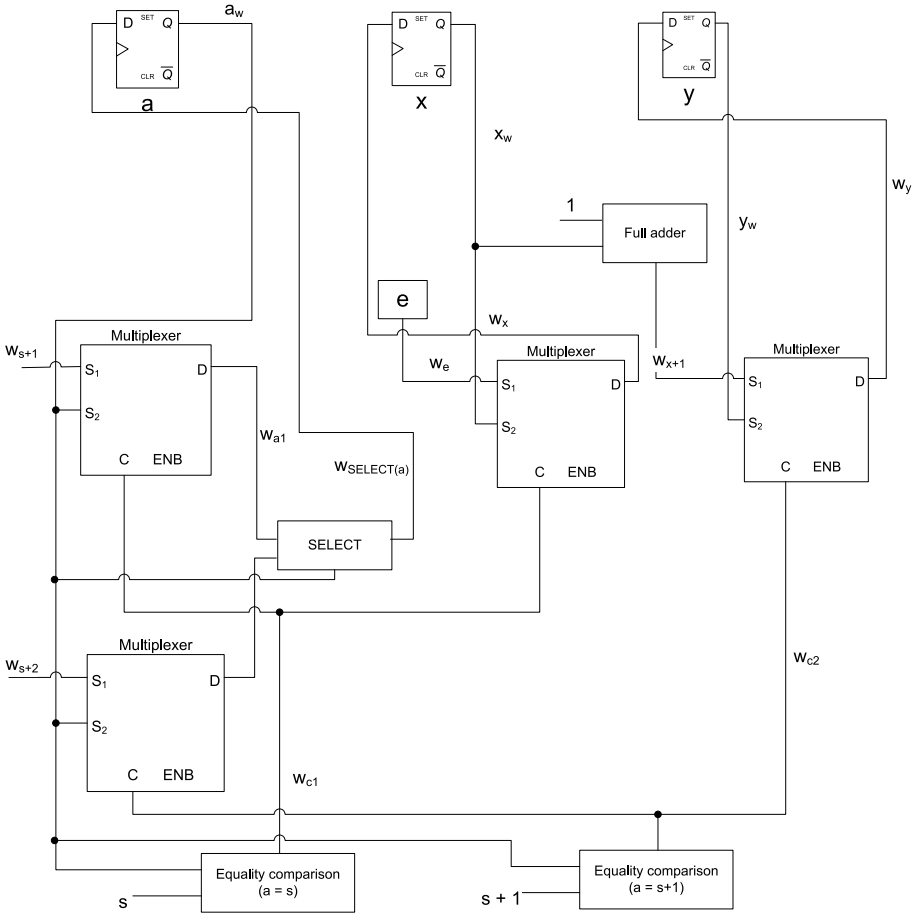

Figure 5.6: Hardware output for the program $x \underset{\text{snc}}{:=} e \fatsemi y \underset{\text{snc}}{:=} x + 1$

- **Second normal form**. The main reason for introducing our second normal form is to simplify the parallelism from the first normal form and make it amenable for implementation in hardware devices. To achieve this goal, all steps in the first normal are merged into a single computation describing the combinatorial and sequential aspects of an execution's clock-cycle. The different control paths followed by the execution are represented in the second normal form's step by means of conditional expressions and the selection of the right values used to update the program variables.

- **Hardware generation from our second normal form**. Given the high level of detail and high resemblance to hardware of the second normal form, its correct mapping into FPGAs is argued based in the usage of basic, well-understood hardware components: (a) combinatorial logic (adders, subtracters, logic gates and multiplexers) to encode the combinatorial aspects of the step; (b) flip-flops to store the program and control variables; and (c) wires to inter-connect the different blocks.

The next chapter completes our algebraic treatment of Handel-C's compilation by showing how the communication primitives and the **priAlt** construct can be reduced to the same set of normal forms.

Chapter 6

Communications and prioritised choice

"The psychological profiling [of a programmer] is mostly the ability to shift levels of abstraction, from low level to high level. To see something in the small and to see something in the large"

— Donald Knuth

Up to this point, we have addressed the hardware compilation of the C-based aspects of Handel-C together with its parallel construct. In this section we extend our approach to address the missing features from our subset of Handel-C: communications and prioritised choice.

The key to compiling the input, output and **priAlt** constructs with the algebraic approach is to be able to abstract their wire-based implementation details and to decompose them into more primitive operations over channels. We begin this section by showing how the compilation theorems can take advantage of the extended primitives over channels defined in chapter 4 to reduce the communication primitives to first normal form.

To address the reduction of the first normal form encoding of the communication primitives into second normal form we adopt the strategy of re-using the compilation theorems introduced in the previous section. Clearly, the main limitation towards achieving this goal is the fact that the existing compilation theorems can only handle particular forms of guarded multiple assignments. We address this limitation by means of introducing additional normal form transformations that allow us to formulate the communication primitives as assignments guarded by particular conditions.

6.1 First normal form encoding of the communication primitives

The reduction of the basic communication primitives into first normal form takes advantage of the equivalences stated in Laws 4.2.17 and 4.2.18:

$$ch?x = \mu X \bullet \textbf{in-req}(ch); ((x := \textbf{in}(ch))_1 \lhd \textbf{wr}(ch) \rhd \textbf{delay}; X)$$

and

$$ch!e = \mu X \bullet \textbf{out-req}(ch); \textbf{out}(ch, e); (\textbf{delay} \lhd \textbf{rd}(ch) \rhd \textbf{delay}; X)$$

The compilation follows the strategy of encoding the recursive right hand side of the above expression in our first normal form (using the same primitive actions over channels) and then using the equivalences above to establish the link between the normal form and the actual input and output commands.

The compilation strategy for prioritised choice uses a similar strategy but based on the equivalence between the **priAlt** construct and the extended case construct presented in Chapter 4.

6.1.1 Input

The compilation of the input command into first normal form follows the refinement relationship:

$$ch?x \sqsubseteq a : [a = s, (a = s \underset{\text{snc}}{\rightarrow} (\textbf{in-req}(ch); x, a \underset{\text{snc}}{:=} (f, \textbf{in}(ch)) \lhd \textbf{wr}(ch) \rhd (s, x))_1), a = f]$$

As in the case of the compilation of sequential composition into first normal form, we isolate particular cases in the execution of the input command within simpler lemmas that allows us to provide an elegant structure to the proof of its reduction theorem into normal form.

Our first lemma states the relationship between the abstract effects of a successful input command and the execution of our proposed normal form in the same situation. The normal form is a refinement of the execution of the input command due to the presence of control variables and the values they need to hold while the normal form is in operation.

Lemma 6.1.1. *Input normal form simplification 1*

$$\begin{aligned}
&(\textbf{\textit{wr}}(ch))^\top; \textbf{\textit{in-req}}(ch); (x \underset{snc}{:=} \textbf{\textit{in}}(ch))_1 \sqsubseteq \\
&(\textbf{\textit{wr}}(ch))^\top; \\
&\quad a : [a = s, (s = a \underset{snc}{\rightarrow} \textbf{\textit{in-req}}(ch); x, a \underset{snc}{:=} (f \lhd \textbf{\textit{wr}}(ch) \rhd s), (\textbf{\textit{in}}(ch) \lhd \textbf{\textit{wr}}(ch) \rhd x)_1), a = f]
\end{aligned}$$

Proof.

Let $P = (s = a \underset{\text{snc}}{\rightarrow} \textbf{in-req}(ch); x, a \underset{\text{snc}}{:=} (f \lhd \textbf{wr}(ch) \rhd s), (\textbf{in}(ch) \lhd \textbf{wr}(ch) \rhd x)_1)$ in:

$$\begin{aligned}
&(\textbf{wr}(ch))^\top; a : [a = s, (s = a \underset{\text{snc}}{\rightarrow} P), a = f] \\
&= \{\text{Definitions 5.1.17, law 3.3.100}\} \\
&(\textbf{wr}(ch))^\top_{\mathsf{S}}; \mathtt{var}\ a; (a = s)^\top_{\mathsf{S}}; P; *(a = s \underset{\text{snc}}{\rightarrow} P); (a = f)^\bot_{\mathsf{S}}; \mathsf{end}\ a \\
&= \{\text{Expansion of } P \text{ and laws 3.3.92, 3.3.46, 4.2.19}\}
\end{aligned}$$

$$
\begin{array}{l}
\quad \mathtt{var}\ a; (a = s)^{\top}_{\mathsf{S}}; \mathbf{in\text{-}req}(ch); (\mathbf{wr}(ch))^{\top}_{\mathsf{S}}; (a, x \mathrel{:=}_{\mathrm{snc}} f \lhd \mathbf{wr}(ch) \rhd s, \mathbf{in}(ch) \lhd \mathbf{wr}(ch) \rhd x)_1 ; \\
\quad * (a = s \rightarrow_{\mathrm{snc}} P); (a = f)^{\bot}_{\mathsf{S}}; \mathsf{end}\ a \\
= \{\text{Laws 3.3.58, 3.3.92, 3.3.46, 4.2.19}\} \\
\quad (\mathbf{wr}(ch))^{\top}_{\mathsf{S}}; \mathtt{var}\ a; (a = s)^{\top}_{\mathsf{S}}; \mathbf{in\text{-}req}(ch); (a, x \mathrel{:=}_{\mathrm{snc}} f, \mathbf{in}(ch))_1 ; *(a = s \rightarrow_{\mathrm{snc}} P); (a = f)^{\bot}_{\mathsf{S}}; \mathsf{end}\ a \\
= \{\text{Law 4.2.6}\} \\
\quad (\mathbf{wr}(ch))^{\top}_{\mathsf{S}}; \mathtt{var}\ a; (a = s)^{\top}_{\mathsf{S}}; \mathbf{in\text{-}req}(ch); \\
\quad (a, x \mathrel{:=}_{\mathrm{snc}} f, \mathbf{in}(ch))_1; (a = f)^{\bot}_{\mathsf{S}}; *(a = s \rightarrow_{\mathrm{snc}} P); (a = f)^{\bot}_{\mathsf{S}}; \mathsf{end}\ a \\
= \{\text{Laws 3.3.99, 3.3.49 and 4.2.6}\} \\
\quad (\mathbf{wr}(ch))^{\top}_{\mathsf{S}}; \mathtt{var}\ a; (a = s)^{\top}_{\mathsf{S}}; \mathbf{in\text{-}req}(ch); (a, x \mathrel{:=}_{\mathrm{snc}} f, \mathbf{in}(ch))_1; \mathsf{end}\ a \\
= \{\text{Law 3.3.72}\} \\
\quad (\mathbf{wr}(ch))^{\top}_{\mathsf{S}}; \mathtt{var}\ a; (a = s)^{\top}_{\mathsf{S}}; \mathbf{in\text{-}req}(ch); (x \mathrel{:=}_{\mathrm{snc}} \mathbf{in}(ch))_1 ; \mathsf{end}\ a \\
\sqsupseteq \{\text{Laws 3.3.40, 3.3.94, 3.3.70}\} \\
\quad (\mathbf{wr}(ch))^{\top}_{\mathsf{S}}; \mathbf{in\text{-}req}(ch); (x \mathrel{:=}_{\mathrm{snc}} \mathbf{in}(ch))_1 \qquad \Box
\end{array}
$$

When the communication is not possible, the effects of the input command during the first clock cycle followed by our input normal form are refined by the execution of the input normal form in the same context. Again, the relationship is refinement rather than equality due to the fact that the left-hand side of our lemma does not restrict the value of the control variable a during the first clock cycle.

Lemma 6.1.2. *Input normal form simplification 2*

$$
\begin{array}{l}
(\neg \boldsymbol{wr}(ch))^{\top}; \boldsymbol{in\text{-}req}(ch); (x \mathrel{:=}_{\boldsymbol{snc}} x)_1; \\
\quad a : [a = s, (s = a \rightarrow_{\boldsymbol{snc}} \boldsymbol{in\text{-}req}(ch); x, a \mathrel{:=}_{\boldsymbol{snc}} (f \lhd \boldsymbol{wr}(ch) \rhd s), (\boldsymbol{in}(ch) \lhd \boldsymbol{wr}(ch) \rhd x)_1), a = f] \sqsubseteq \\
(\neg \boldsymbol{wr}(ch))^{\top}; \\
\quad a : [a = s, (s = a \rightarrow_{\boldsymbol{snc}} \boldsymbol{in\text{-}req}(ch); x, a \mathrel{:=}_{\boldsymbol{snc}} (f \lhd \boldsymbol{wr}(ch) \rhd s), (\boldsymbol{in}(ch) \lhd \boldsymbol{wr}(ch) \rhd x)_1), a = f]
\end{array}
$$

Proof.

Let $(P = s = a \rightarrow_{\mathrm{snc}} \mathbf{in\text{-}req}(ch); x, a \mathrel{:=}_{\mathrm{snc}} (f \lhd \mathbf{wr}(ch) \rhd s), (\mathbf{in}(ch) \lhd \mathbf{wr}(ch) \rhd x)_1)$ in:

$$
\begin{array}{l}
\quad (\neg \mathbf{wr}(ch))^{\top}_{\mathsf{S}}; a : [a = s, (a = s \rightarrow_{\mathrm{snc}} P), a = f] \\
= \{\text{Definitions 5.1.17, law 3.3.100}\} \\
\quad (\neg \mathbf{wr}(ch))^{\top}_{\mathsf{S}}; \mathtt{var}\ c; (c = s)^{\top}_{\mathsf{S}}; P; *(c = s \rightarrow_{\mathrm{snc}} P); (c = f)^{\bot}_{\mathsf{S}}; \mathsf{end}\ c
\end{array}
$$

$$
\begin{array}{l}
= \{\text{Value of } P \text{ and laws 3.3.92, 3.3.46, 4.2.19}\} \\
\quad \mathtt{var}\ c; (c = s)^{\top}_{\mathsf{S}}; \mathbf{in\text{-}req}(ch); (\neg \mathbf{wr}(ch))^{\top}_{\mathsf{S}}; (c, x \mathrel{:=}_{\mathrm{snc}} f \lhd \mathbf{wr}(ch) \rhd s, \mathbf{in}(ch) \lhd \mathbf{wr}(ch) \rhd x)_1 ; \\
\quad * (c = s \overrightarrow{\ }_{\mathrm{snc}} P); (c = f)^{\bot}_{\mathsf{S}}; \mathsf{end}\ c \\
= \{\text{Laws 3.3.58, 3.3.92, 3.3.46, 4.2.19}\} \\
\quad (\neg \mathbf{wr}(ch))^{\top}_{\mathsf{S}}; \mathtt{var}\ c; (c = s)^{\top}_{\mathsf{S}}; \mathbf{in\text{-}req}(ch); (c, x \mathrel{:=}_{\mathrm{snc}} s, x)_1 ; *(c = s \overrightarrow{\ }_{\mathrm{snc}} P); (c = f)^{\bot}_{\mathsf{S}}; \mathsf{end}\ c \\
\sqsupseteq \{\text{Laws 4.2.6, 3.3.40, 3.3.92}\} \\
\quad (\mathbf{wr}(ch))^{\top}_{\mathsf{S}}; \mathbf{in\text{-}req}(ch); \mathtt{var}\ c; (c, x \mathrel{:=}_{\mathrm{snc}} f, x)_1; (c = s)^{\bot}_{\mathsf{S}}; *(c = s \overrightarrow{\ }_{\mathrm{snc}} P); (c = f)^{\bot}_{\mathsf{S}}; \mathsf{end}\ c \\
\sqsupseteq \{\text{Laws 3.3.73 and 3.3.94}\} \\
\quad (\mathbf{wr}(ch))^{\top}_{\mathsf{S}}; \mathbf{in\text{-}req}(ch); (x \mathrel{:=}_{\mathrm{snc}} x)_1; \mathtt{var}\ c; (c = s)^{\bot}_{\mathsf{S}}; *(c = s \overrightarrow{\ }_{\mathrm{snc}} P); (c = f)^{\bot}_{\mathsf{S}}; \mathsf{end}\ c \\
= \{\text{Definition 5.1.17}\} \\
\quad (\neg \mathbf{wr}(ch))^{\top}_{\mathsf{S}}; a : [a = s, (s = a \overrightarrow{\ }_{\mathrm{snc}} P), a = f] \qquad \square
\end{array}
$$

With the results above we can easily prove the reduction theorem for the input construct to first normal form.

Theorem 6.1.3. *Input normal form*

$$ch?x \sqsubseteq a : [a = s, (a = s \overrightarrow{\ }_{snc} (\textit{in-req}(ch); x, a \mathrel{:=}_{snc} (f, \textit{in}(ch)) \lhd \textit{wr}(ch) \rhd (s, x))_1), a = f]$$

Proof.

$$
\begin{array}{l}
\quad RHS = RHS \\
\equiv \{\text{Laws 3.3.23 and 3.3.57}\} \\
\quad RHS = ((\mathbf{wr}(ch))^{\top}_{\mathsf{S}}; RHS) \lhd \mathbf{wr}(ch) \rhd ((\neg \mathbf{wr}(ch))^{\top}_{\mathsf{S}}; RHS) \\
\equiv \{\text{Lemmas 6.1.1 and 6.1.2, then law 3.3.57}\} \\
\quad RHS \sqsupseteq (\mathbf{in\text{-}req}(ch); (x \mathrel{:=}_{\mathrm{snc}} \mathbf{in}(ch))_1) \lhd \mathbf{wr}(ch) \rhd (\mathbf{in\text{-}req}(ch); (x \mathrel{:=}_{\mathrm{snc}} x)_1; RHS) \\
\equiv \{\text{Law 4.2.22}\} \\
\quad RHS \sqsupseteq \mathbf{in\text{-}req}(ch); (x \mathrel{:=}_{\mathrm{snc}} \mathbf{in}(ch))_1 \lhd \mathbf{wr}(ch) \rhd ((x \mathrel{:=}_{\mathrm{snc}} x)_1; RHS) \\
\Rightarrow \{\text{Law 2.3.81}\} \\
\quad RHS \sqsupseteq \mu X \bullet \mathbf{in\text{-}req}(ch); (x \mathrel{:=}_{\mathrm{snc}} \mathbf{in}(ch))_1 \lhd \mathbf{wr}(ch) \rhd ((x \mathrel{:=}_{\mathrm{snc}} x)_1; X) \\
\equiv \{\text{Law 4.2.17}\} \\
\quad RHS \sqsupseteq LHS \qquad \square
\end{array}
$$

6.1.2 Output

The compilation of the output command follows the same strategy we used for the input command. The main reduction theorem that establishes the refinement relationship between the output

command and its first-normal form formulation is stated as follows:

$$ch!e \sqsubseteq a : [a = s, (a = s \overrightarrow{}_{\text{snc}} \textbf{out-req}(ch); \textbf{out}(ch, e); (v, a \mathrel{:=}_{\text{snc}} v, (f \lhd \textbf{rd}(ch) \rhd s))_1), a = f]$$

Our first result is the output counterpart of Lemma 6.1.1 and it describes the refinement induced by the normal form encoding of the input command in the context of successful communication.

Lemma 6.1.4. *Output normal form simplification 1*

$$\begin{aligned}
&(\textbf{\textit{rd}}(ch))^{\top}_{S}; \textbf{\textit{out-req}}(ch); \textbf{\textit{out}}(ch, e); \textbf{\textit{delay}} \sqsubseteq \\
&(\textbf{\textit{rd}}(ch))^{\top}_{S}; \\
&\quad a : [a = s, (a = s \overrightarrow{}_{snc} \textbf{\textit{out-req}}(ch); \textbf{\textit{out}}(ch, e); (v, a \mathrel{:=}_{snc} v, (f \lhd \textbf{\textit{rd}}(ch) \rhd s))_1), a = f]
\end{aligned}$$

Proof.

Let $P = \left(\textbf{out-req}(ch); \textbf{out}(ch, e); (a, x \mathrel{:=}_{\text{snc}} f \lhd \textbf{rd}(ch) \rhd s, x)_1\right)$ in:

$$\begin{aligned}
&(\textbf{rd}(ch))^{\top}_{S}; a : [a = s, (a = s \overrightarrow{}_{\text{snc}} P), a = f] \\
&= \{\text{Definitions 5.1.17, law 3.3.100}\} \\
&\quad (\textbf{rd}(ch))^{\top}_{S}; \texttt{var}\ a; (a = s)^{\top}_{S}; P; *(a = s \overrightarrow{}_{\text{snc}} P); (a = f)^{\bot}_{S}; \texttt{end}\ a \\
&= \{\text{Value of } P \text{ and laws 3.3.92, 3.3.46, 4.2.20}\} \\
&\quad \texttt{var}\ a; (a = s)^{\top}_{S}; \textbf{out-req}(ch); (\textbf{rd}(ch))^{\top}_{S}; (a, x \mathrel{:=}_{\text{snc}} f \lhd \textbf{rd}(ch) \rhd s, x)_1 ; \\
&\qquad * (a = s \overrightarrow{}_{\text{snc}} P); (a = f)^{\bot}_{S}; \texttt{end}\ a \\
&= \{\text{3.3.58, 3.3.92, 3.3.46, 4.2.19}\} \\
&\quad (\textbf{rd}(ch))^{\top}_{S}; \texttt{var}\ a; (a = s)^{\top}_{S}; \textbf{out-req}(ch); (a, x \mathrel{:=}_{\text{snc}} f, x)_1 ; *(a = s \overrightarrow{}_{\text{snc}} P); (a = f)^{\bot}_{S}; \texttt{end}\ a \\
&= \{\text{Law 4.2.6}\} \\
&\quad (\textbf{rd}(ch))^{\top}_{S}; \texttt{var}\ a; (a = s)^{\top}_{S}; \textbf{out-req}(ch); \\
&\quad (a, x \mathrel{:=}_{\text{snc}} f, x)_1; (a = f)^{\bot}_{S}; *(a = s \overrightarrow{}_{\text{snc}} P); (a = f)^{\bot}_{S}; \texttt{end}\ a \\
&= \{\text{Laws 3.3.99, 3.3.49 and 4.2.6}\} \\
&\quad (\textbf{rd}(ch))^{\top}_{S}; \texttt{var}\ a; (a = s)^{\top}_{S}; \textbf{out-req}(ch); (a, x \mathrel{:=}_{\text{snc}} f, x)_1; \texttt{end}\ a \\
&= \{\text{Law 3.3.72}\} \\
&\quad (\textbf{rd}(ch))^{\top}_{S}; \texttt{var}\ a; (a = s)^{\top}_{S}; \textbf{out-req}(ch); (x \mathrel{:=}_{\text{snc}} x)_1 ; \texttt{end}\ a \\
&\sqsupseteq \{\text{Laws 3.3.40, 3.3.94 and 3.3.70, theorem 4.2.2}\} \\
&\quad (\textbf{rd}(ch))^{\top}_{S}; \textbf{out-req}(ch); \textbf{delay}
\end{aligned}$$

□

Similarly to Lemma 6.1.2, the following result captures the relationship between the output command and our normal form encoding in the context where the communication is not possible during the first clock cycle.

Lemma 6.1.5. *Output normal form simplification 2*

$$
\begin{array}{l}
(\neg \mathbf{rd}(ch))^{\top}_{\mathsf{S}}; \mathbf{out\text{-}req}(ch); \mathbf{out}(ch, e); \mathbf{delay}; \\
\quad a : [a = s, (a = s \overrightarrow{\ }_{snc} \mathbf{out\text{-}req}(ch); \mathbf{out}(ch, e); (v, a :=_{snc} v, (f \lhd \mathbf{rd}(ch) \rhd s))_1), a = f] \sqsubseteq \\
(\neg \mathbf{rd}(ch))^{\top}_{\mathsf{S}}; \\
\quad a : [a = s, (a = s \overrightarrow{\ }_{snc} \mathbf{out\text{-}req}(ch); \mathbf{out}(ch, e); (v, a :=_{snc} v, (f \lhd \mathbf{rd}(ch) \rhd s))_1), a = f]
\end{array}
$$

Proof.

Let $\left(P = \mathbf{out\text{-}req}(ch); \mathbf{out}(ch, e); (a, x :=_{\mathrm{snc}} f \lhd \mathbf{rd}(ch) \rhd s, x)_1\right)$ in:

$$
\begin{array}{ll}
 & (\neg \mathbf{rd}(ch))^{\top}_{\mathsf{S}}; a : [a = s, (a = s \overrightarrow{\ }_{\mathrm{snc}} P), a = f] \\
= & \{\text{Definitions 5.1.17, law 3.3.100}\} \\
 & (\neg \mathbf{rd}(ch))^{\top}_{\mathsf{S}}; \mathtt{var}\ a; (a = s)^{\top}_{\mathsf{S}}; P; *(a = s \overrightarrow{\ }_{\mathrm{snc}} P); (a = f)^{\perp}_{\mathsf{S}}; \mathsf{end}\ a \\
= & \{\text{Expansion of } P \text{ and laws 3.3.92, 3.3.46, 4.2.20 and 4.2.21}\} \\
 & \mathtt{var}\ a; (a = s)^{\top}_{\mathsf{S}}; \mathbf{out\text{-}req}(ch); (\neg \mathbf{rd}(ch))^{\top}_{\mathsf{S}}; \mathbf{out}(ch, e); (\neg \mathbf{rd}(ch))^{\top}_{\mathsf{S}}; \\
 & \quad (a, x :=_{\mathrm{snc}} f \lhd \mathbf{rd}(ch) \rhd s, x)_1; *(a = s \overrightarrow{\ }_{\mathrm{snc}} P); (a = f)^{\perp}_{\mathsf{S}}; \mathsf{end}\ a \\
= & \{\text{Laws 3.3.58, 3.3.92, 3.3.46, 4.2.19 and 4.2.21}\} \\
 & (\neg \mathbf{rd}(ch))^{\top}_{\mathsf{S}}; \mathtt{var}\ a; (a = s)^{\top}_{\mathsf{S}}; \mathbf{out\text{-}req}(ch); \mathbf{out}(ch, e); (a, x :=_{\mathrm{snc}} s, x)_1; \\
 & \quad * (a = s \overrightarrow{\ }_{\mathrm{snc}} P); (a = f)^{\perp}_{\mathsf{S}}; \mathsf{end}\ a \\
\sqsupseteq & \{\text{Laws 4.2.6 and 3.3.40}\} \\
 & (\mathbf{rd}(ch))^{\top}_{\mathsf{S}}; \mathbf{out\text{-}req}(ch); \mathbf{out}(ch, e); \mathtt{var}\ a; (a, x :=_{\mathrm{snc}} f, x)_1; \\
 & \quad (a = s)^{\perp}_{\mathsf{S}}; *(a = s \overrightarrow{\ }_{\mathrm{snc}} P); (a = f)^{\perp}_{\mathsf{S}}; \mathsf{end}\ a \\
\sqsupseteq & \{\text{Laws 3.3.73 and 3.3.94}\} \\
 & (\mathbf{rd}(ch))^{\top}_{\mathsf{S}}; \mathbf{out\text{-}req}(ch); (x :=_{\mathrm{snc}} x)_1; \mathtt{var}\ a; (a = s)^{\perp}_{\mathsf{S}}; *(a = s \overrightarrow{\ }_{\mathrm{snc}} P); (a = f)^{\perp}_{\mathsf{S}}; \mathsf{end}\ a \\
= & \{\text{Theorem 4.2.2, law 3.3.9 and definition 5.1.17}\} \\
 & (\neg \mathbf{rd}(ch))^{\top}_{\mathsf{S}}; \mathbf{out\text{-}req}(ch); \mathbf{out}(ch, e); \mathbf{delay}; a : [a = s, (a = s \overrightarrow{\ }_{\mathrm{snc}} P), a = f] \qquad \square
\end{array}
$$

The theorem describing the first normal form encoding of the output construct can be proved from the above results as follows:

Theorem 6.1.6. *Output normal form*

$$ch!e \sqsubseteq a : [a = s, (a = s \overrightarrow{\ }_{snc} \mathbf{out\text{-}req}(ch); \mathbf{out}(ch, e); (v, a :=_{snc} v, (f \lhd \mathbf{rd}(ch) \rhd s))_1), a = f]$$

Proof.

$$
\begin{array}{ll}
& RHS = RHS \\
\equiv & \{\text{Laws 3.3.23 and 3.3.57}\} \\
& RHS = ((\mathbf{wr}(ch))^{\top}_{\mathsf{S}}; RHS) \lhd \mathbf{wr}(ch) \rhd ((\neg \mathbf{wr}(ch))^{\top}_{\mathsf{S}}; RHS) \\
\equiv & \{\text{Lemmas 6.1.4 and 6.1.5, then law 3.3.57}\} \\
& RHS \sqsupseteq (\mathbf{out\text{-}req}(ch); \mathbf{out}(ch, e); \mathbf{delay}) \lhd \mathbf{wr}(ch) \rhd (\mathbf{out\text{-}req}(ch); \mathbf{out}(ch, e); \mathbf{delay}; RHS) \\
\equiv & \{\text{Laws 4.2.23 and 4.2.24}\} \\
& RHS \sqsupseteq \mathbf{out\text{-}req}(ch); \mathbf{out}(ch, e); (\mathbf{delay} \lhd \mathbf{wr}(ch) \rhd \mathbf{delay}; RHS) \\
\Rightarrow & \{\text{Law 2.3.81}\} \\
& RHS \sqsupseteq \mu X \bullet \mathbf{out\text{-}req}(ch); \mathbf{out}(ch, e); (\mathbf{delay} \lhd \mathbf{wr}(ch) \rhd \mathbf{delay}; X) \\
\equiv & \{\text{Law 4.2.17}\} \\
& RHS \sqsupseteq LHS
\end{array}
$$

□

6.1.3 Default-clause prioritised choice priAlt

We begin by tackling the compilation of the binary version of the **priAlt** construct with default clause. As the n-way **priAlt** construct with default clause can be expressed as nested binary **priAlt**s (law 4.1.24), this result is enough to translate any **priAlt** with a default clause to first normal form.

Our compilation strategy consists of generating a normal form where the first step contains a case statement like the one described in Chapter 4. The case construct attempts the communication for each alternative in a waterfall fashion. If either of the alternatives becomes **true**, the case statement also activates the actions that correspond to the first clock cycle of execution of that alternative. The rest of the normal form corresponds to steps associated to the sub-programs within each case. More formally:

$$
\mathbf{priAlt}\left\{\begin{array}{l}\mathbf{case}\ g\mathbf{:}\ a : [s_1, \left(s_1 \underset{\text{snc}}{\rightarrow} P_1 \parallel_{\hat{M}} b_2 \underset{\text{snc}}{\rightarrow} R\right), f] \fatsemi \mathbf{break}; \\ \mathbf{default:}\ a : [s_2, \left(s_2 \underset{\text{snc}}{\rightarrow} P_2 \parallel_{\hat{M}} b_3 \underset{\text{snc}}{\rightarrow} P_3\right), f]\end{array}\right\} \sqsubseteq
$$

$$
a : [s, \left(\begin{array}{l} s \underset{\text{snc}}{\rightarrow} (\mathbf{case}\ req(g); chk(g)\ ?\ (act(g) \parallel (a \underset{\text{snc}}{:=} s_1)_1) \mid P_2) \\ \parallel_{\hat{M}} s_1 \underset{\text{snc}}{\rightarrow} P_1 \parallel_{\hat{M}} b_2 \underset{\text{snc}}{\rightarrow} R \parallel_{\hat{M}} s_2 \underset{\text{snc}}{\rightarrow} P_2 \parallel_{\hat{M}} b_3 \underset{\text{snc}}{\rightarrow} P_3 \end{array}\right), f]
$$

The proof of this result is structured following the same outline we have used to prove the basic communication primitives. We begin by showing the effect of executing our normal form encoding of the **priAlt** construct when the first communication is possible.

Lemma 6.1.7.

$$(chk(g))^{\top}; req(g); act(g); a : [s_1, \left(s_1 \overrightarrow{}_{snc} P_1 \parallel_{\hat{M}} b_2 \overrightarrow{}_{snc} R\right), f] \sqsubseteq$$
$$(chk(g))^{\top}; a : [s, \left(\begin{array}{l} s \overrightarrow{}_{snc} (\textbf{\textit{case}}\ req(g); chk(g)\ ?\ \left(act(g) \parallel_{\hat{M}} (c :=_{snc} s_1)_1\right) \mid P_2) \\ \parallel_{\hat{M}} s_1 \overrightarrow{}_{snc} P_1 \parallel_{\hat{M}} b_2 \overrightarrow{}_{snc} R \parallel_{\hat{M}} s_2 \overrightarrow{}_{snc} P_2 \parallel_{\hat{M}} b_3 \overrightarrow{}_{snc} P_3 \end{array} \right), f]$$

Proof.

$$\begin{array}{ll} & RHS \\ \sqsupseteq & \{\text{Lemma 5.1.22, definition 5.1.17, laws 3.3.100 and 3.3.40}\} \\ & (chk(g))^{\top}_{S}; \mathtt{var}\ a; (\textbf{case}\ req(g); chk(g)\ ?\ \left(act(g) \parallel_{\hat{M}} (a :=_{snc} s_1)_1\right) \mid P_2); \\ & *\left(s_1 \overrightarrow{}_{snc} P_1 \parallel_{\hat{M}} b_2 \overrightarrow{}_{snc} R\right); (a = f)^{\perp}_{S}; \mathsf{end}\ a \\ = & \{\text{Laws 3.3.92 then definition 4.2.34 followed by law 4.2.43}\} \\ & \mathtt{var}\ a; req(g); (chk(g))^{\top}_{S}; \\ & \left((act(g) \parallel_{\hat{M}} (a :=_{snc} s_1)_1) \lhd chk(g) \rhd P_2\right); *\left(s_1 \overrightarrow{}_{snc} P_1 \parallel_{\hat{M}} b_2 \overrightarrow{}_{snc} R\right); (a = f)^{\perp}_{S}; \mathsf{end}\ a \\ = & \{\text{Laws 3.3.58, 4.2.5,3.3.92 and 4.2.43}\} \\ & (chk(g))^{\top}_{S}; req(g); \mathtt{var}\ a; \\ & (act(g) \parallel_{\hat{M}} (a :=_{snc} s_1)_1); (a = s_1)^{\top}_{S}; *\left(s_1 \overrightarrow{}_{snc} P_1 \parallel_{\hat{M}} b_2 \overrightarrow{}_{snc} R\right); (a = f)^{\perp}_{S}; \mathsf{end}\ a \\ \sqsupseteq & \{\text{Laws 3.3.73 and 3.3.92}\} \\ & (chk(g))^{\top}_{S}; req(g); act(g); \mathtt{var}\ a; (a = s_1)^{\top}_{S}; *\left(s_1 \overrightarrow{}_{snc} P_1 \parallel_{\hat{M}} b_2 \overrightarrow{}_{snc} R\right); (a = f)^{\perp}_{S}; \mathsf{end}\ a \\ = & \{\text{Definition 5.1.17}\} \\ & LHS \end{array}$$

□

Similarly, when the first alternative cannot be activated, the default clause gets triggered.

Lemma 6.1.8.

$$(\neg chk(g))^{\top}_{S}; req(g); a : [s_2, \left(s_2 \overrightarrow{}_{snc} P_2 \parallel_{\hat{M}} b_3 \overrightarrow{}_{snc} P_3\right), f] \sqsubseteq$$
$$(\neg chk(g))^{\top}_{S}; a : [s, \left(\begin{array}{l} s \overrightarrow{}_{snc} (\textbf{\textit{case}}\ req(g); chk(g)\ ?\ \left(act(g) \parallel_{\hat{M}} (a :=_{snc} s_1)_1\right) \mid P_2) \\ \parallel_{\hat{M}} s_1 \overrightarrow{}_{snc} P_1 \parallel_{\hat{M}} b_2 \overrightarrow{}_{snc} R \parallel_{\hat{M}} s_2 \overrightarrow{}_{snc} P_2 \parallel_{\hat{M}} b_3 \overrightarrow{}_{snc} P_3 \end{array} \right), f]$$

Proof.

Let $G_1 = (s \overrightarrow{}_{snc} P_2)$ and $G_2 = \left(s_2 \overrightarrow{}_{snc} P_2 \parallel_{\hat{M}} b_3 \overrightarrow{}_{snc} P_3\right)$ in:

$$(\neg chk(g))^{\top}_{S}; a : [s, \left(\begin{array}{l} s \overrightarrow{}_{snc} (\textbf{case}\ req(g); chk(g)\ ?\ \left(act(g) \parallel_{\hat{M}} (a :=_{snc} s_1)_1\right) \mid P_2) \\ \parallel_{\hat{M}} s_1 \overrightarrow{}_{snc} P_1 \parallel_{\hat{M}} b_2 \overrightarrow{}_{snc} R \parallel_{\hat{M}} s_2 \overrightarrow{}_{snc} P_2 \parallel_{\hat{M}} b_3 \overrightarrow{}_{snc} P_3 \end{array} \right), f]$$

$\sqsupseteq$ {Lemma 5.1.22, definition 5.1.17 and law 3.3.100}

$(\neg chk(g))^{\top}_{\mathsf{S}}; \mathtt{var}\ a; (a = s)^{\top}_{\mathsf{S}}(\mathbf{case}\ req(g); chk(g)\ ?\ (act(g) \parallel (a \underset{\text{snc}}{:=} s_1)_1) \mid P_2);$

$* \left(s_2 \underset{\text{snc}}{\rightarrow} P_2 \parallel_{\hat{M}} b_3 \underset{\text{snc}}{\rightarrow} P_3\right); (a = f)^{\perp}_{\mathsf{S}}; \mathsf{end}\ a$

= {Laws 3.3.92, 3.3.46, then definition 4.2.34 followed by law 4.2.43}

$\mathtt{var}\ a; req(g); (a = s)^{\top}_{\mathsf{S}}; (\neg chk(g))^{\top}_{\mathsf{S}}; ((act(g) \parallel (a \underset{\text{snc}}{:=} s_1)_1) \lhd chk(g) \rhd P_2);$

$* \left(s_2 \underset{\text{snc}}{\rightarrow} P_2 \parallel_{\hat{M}} b_3 \underset{\text{snc}}{\rightarrow} P_3\right); (a = f)^{\perp}_{\mathsf{S}}; \mathsf{end}\ a$

= {Laws 3.3.22, 3.3.58, 4.2.5, 3.3.92 and 4.2.43}

$(\neg chk(g))^{\top}_{\mathsf{S}}; req(g); \mathtt{var}\ a; (a = s)^{\top}_{\mathsf{S}}; P_2; * \left(s_2 \underset{\text{snc}}{\rightarrow} P_2 \parallel_{\hat{M}} b_3 \underset{\text{snc}}{\rightarrow} P_3\right); (a = f)^{\perp}_{\mathsf{S}}; \mathsf{end}\ a$

= {Law 3.3.45, definition 3.3.149}

$(\neg chk(g))^{\top}_{\mathsf{S}}; req(g); \mathtt{var}\ a; (a = s)^{\top}_{\mathsf{S}}; s \underset{\text{snc}}{\rightarrow} P_2; * \left(s_2 \underset{\text{snc}}{\rightarrow} P_2 \parallel_{\hat{M}} b_3 \underset{\text{snc}}{\rightarrow} P_3\right); (a = f)^{\perp}_{\mathsf{S}}; \mathsf{end}\ a$

= {$s \notin cont(G_2) \Rightarrow inh(G_1, G_1) \wedge inh(G_2, G_1)$ then law 5.1.15, definition 5.1.17}

$(\neg chk(g))^{\top}_{\mathsf{S}}; req(g); a : [s, \left(s \underset{\text{snc}}{\rightarrow} P_2 \parallel_{\hat{M}} s_2 \underset{\text{snc}}{\rightarrow} P_2 \parallel_{\hat{M}} b_3 \underset{\text{snc}}{\rightarrow} P_3\right), f]$

$\sqsupseteq$ {Lemma 5.1.24}

$(\neg chk(g))^{\top}_{\mathsf{S}}; req(g); a : [s_2, \left(s_2 \underset{\text{snc}}{\rightarrow} P_2 \parallel_{\hat{M}} b_3 \underset{\text{snc}}{\rightarrow} P_3\right), f]$ □

With the above results we are now in condition to prove the compilation theorem that states how the **priAlt** construct gets reduced to first normal form.

Theorem 6.1.9. *Normal form encoding of* ***priAlt*** *(with default clause)*

$$\textbf{\textit{priAlt}} \left\{ \begin{array}{l} \textbf{\textit{case}}\ g\textbf{:}\ a : [s_1, \left(s_1 \underset{snc}{\rightarrow} P_1 \parallel_{\hat{M}} b_2 \underset{snc}{\rightarrow} R\right), f]\ ;\!\!\!\!\circ\ \textbf{\textit{break}}; \\ \textbf{\textit{default:}}\ a : [s_2, \left(s_2 \underset{snc}{\rightarrow} P_2 \parallel_{\hat{M}} b_3 \underset{snc}{\rightarrow} P_3\right), f] \end{array} \right\} \sqsubseteq$$

$$a : [s, \left(\begin{array}{l} s \underset{snc}{\rightarrow} (\textbf{\textit{case}}\ req(g); chk(g)\ ?\ \left(act(g) \parallel_{\hat{M}} (a \underset{\text{snc}}{:=} s_1)_1\right) \mid P_2) \\ \parallel_{\hat{M}} s_1 \underset{snc}{\rightarrow} P_1 \parallel_{\hat{M}} b_2 \underset{snc}{\rightarrow} R \parallel_{\hat{M}} s_2 \underset{snc}{\rightarrow} P_2 \parallel_{\hat{M}} b_3 \underset{snc}{\rightarrow} P_3 \end{array} \right), f]$$

Proof.

RHS

= {Laws 3.3.23, 3.3.57}

$((chk(g))^{\top}_{\mathsf{S}}; RHS) \lhd chk(g) \rhd ((\neg chk(g))^{\top}_{\mathsf{S}}; RHS)$

$\sqsupseteq$ {Lemmas 6.1.7 and 6.1.8, then law 3.3.57}

$(req(g); act(g); a : [s_1, \left(s_1 \underset{\text{snc}}{\rightarrow} P_1 \parallel_{\hat{M}} b_2 \underset{\text{snc}}{\rightarrow} R\right), f]) \lhd chk(g) \rhd$

$(req(g); a : [s_2, \left(s_2 \underset{\text{snc}}{\rightarrow} P_2 \parallel_{\hat{M}} b_3 \underset{\text{snc}}{\rightarrow} P_3\right), f])$

= {Law 4.2.44 then definition 4.2.34}

$$\mathbf{case}\left(\begin{array}{l} req(g); chk(g)\ ?\ act(g); a : [s_1, \left(s_1 \overrightarrow{}_{\text{snc}} P_1 \parallel_{\hat{M}} b_2 \overrightarrow{}_{\text{snc}} R\right), f] \mid \\ a : [s_2, \left(s_2 \overrightarrow{}_{\text{snc}} P_2 \parallel_{\hat{M}} b_3 \overrightarrow{}_{\text{snc}} P_3\right), f] \end{array}\right)$$

= {Theorem 4.2.46}

LHS □

6.1.4 Non-default-clause priAlt

As there is no way of expressing the case of the **priAlt** construct without a default clause as a binary operator, we need to provide a general theorem that comprises all cases. Fortunately, the structure for the construct is symmetric, so we will base the proof of the reduction rules on the case of a prialt with two guards. The proof for larger cases follows the same structure.

The normal form encoding of the non-default **priAlt** also uses the case construct to capture the behaviour during the first clock cycle. The main difference with the normal form associated to the default-clause **priAlt** described before is the fact that an additional (unconditional) alternative is added in case all previous alternatives have failed. This additional clause makes the first step repeat itself in the next clock cycle (by means of setting the control variables to the initial state of the normal form). More precisely, the compilation rule is described as follows:

$$\mathbf{priAlt}\left\{\begin{array}{l} \mathbf{case}\ g_1\mathbf{:}\ a : [s_1, \left(s_1 \overrightarrow{}_{\text{snc}} P_1 \parallel_{\hat{M}} b_2 \overrightarrow{}_{\text{snc}} R\right), f] \fatsemi \mathbf{break}; \\ \mathbf{case}\ g_2\mathbf{:}\ a : [s_2, \left(s_2 \overrightarrow{}_{\text{snc}} P_2 \parallel_{\hat{M}} b_3 \overrightarrow{}_{\text{snc}} P_3\right), f] \fatsemi \mathbf{break} \end{array}\right\} \sqsubseteq$$

$$a : [s, \left(\begin{array}{l} s \overrightarrow{}_{\text{snc}} \mathbf{case}\left(\begin{array}{l} req(g); chk(g)\ ?\ \left(act(g) \parallel_{\hat{M}} (a \coloneqq_{\text{snc}} s_1)_1\right) \mid \\ req(g_1); chk(g_1)\ ?\ \left(act(g_1) \parallel_{\hat{M}} (a \coloneqq_{\text{snc}} s_2)_1\right) \mid \\ (a \coloneqq_{\text{snc}} s)_1 \end{array}\right) \\ \parallel_{\hat{M}} s_1 \overrightarrow{}_{\text{snc}} P_1 \parallel_{\hat{M}} b_2 \overrightarrow{}_{\text{snc}} R \parallel_{\hat{M}} s_2 \overrightarrow{}_{\text{snc}} P_2 \parallel_{\hat{M}} b_3 \overrightarrow{}_{\text{snc}} P_3 \end{array}\right), f]$$

We begin by proving similar lemmas to the ones associated to the **priAlt** construct with default clause. The first lemma accounts for the execution of the non-default **priAlt** when the communication inside its first alternative is possible.

Lemma 6.1.10.

$$(chk(g))^{\top}_{S}; req(g); act(g); a : [s_1, \left(s_1 \overrightarrow{}_{snc} P_1 \parallel_{\hat{M}} b_2 \overrightarrow{}_{snc} R\right), f] \sqsubseteq$$

$$(chk(g))^{\top}_{S}; a : [s, \left(\begin{array}{l} s \overrightarrow{}_{\text{snc}} \textit{case}\left(\begin{array}{l} req(g); chk(g)\ ?\ \left(act(g) \parallel_{\hat{M}} (a \coloneqq_{\text{snc}} s_1)_1\right) \mid \\ req(g_1); chk(g_1)\ ?\ \left(act(g_1) \parallel_{\hat{M}} (a \coloneqq_{\text{snc}} s_2)_1\right) \mid \\ (a \coloneqq_{\text{snc}} s)_1 \end{array}\right) \\ \parallel_{\hat{M}} s_1 \overrightarrow{}_{snc} P_1 \parallel_{\hat{M}} b_2 \overrightarrow{}_{snc} R \parallel_{\hat{M}} s_2 \overrightarrow{}_{snc} P_2 \parallel_{\hat{M}} b_3 \overrightarrow{}_{snc} P_3 \end{array}\right), f]$$

Proof.

Similar to Lemma 6.1.7. □

Our next lemma is similar to the previous one but it describes the case where the **priAlt** fails when trying to activate its first case but it succeeds to activate its second alternative.

Lemma 6.1.11.

$$
\begin{array}{l}
(\neg chk(g))^{\top}_{S}; (chk(g_1))^{\top}_{S}; \\
\quad req(g); req(g_1); act(g_1); a : [s_1, \left(s_2 \underset{snc}{\rightarrow} P_2 \parallel_{\hat{M}} b_3 \underset{snc}{\rightarrow} P_3\right), f] \sqsubseteq \\
(\neg chk(g))^{\top}_{S}; (chk(g_1))^{\top}_{S}; \\
\quad a : [s, \left(\begin{array}{l} s \underset{snc}{\rightarrow} \textbf{case} \left(\begin{array}{l} req(g); chk(g) \ ? \ \left(act(g) \parallel_{\hat{M}} (a \underset{snc}{:=} s_1)_1\right) \mid \\ req(g_1); chk(g_1) \ ? \ \left(act(g_1) \parallel_{\hat{M}} (a \underset{snc}{:=} s_2)_1\right) \mid \\ (a \underset{snc}{:=} s)_1 \end{array} \right) \\ \parallel_{\hat{M}} s_1 \underset{snc}{\rightarrow} P_1 \parallel_{\hat{M}} b_2 \underset{snc}{\rightarrow} R \parallel_{\hat{M}} s_2 \underset{snc}{\rightarrow} P_2 \parallel_{\hat{M}} b_3 \underset{snc}{\rightarrow} P_3 \end{array} \right), f]
\end{array}
$$

Proof.

$$
\text{Let } P = \left(\begin{array}{l} s \underset{snc}{\rightarrow} \textbf{case} \left(\begin{array}{l} req(g); chk(g) \ ? \ (act(g) \parallel (a \underset{snc}{:=} s_1)_1) \mid \\ req(g_1); chk(g_1) \ ? \ (act(g_1) \parallel (a \underset{snc}{:=} s_2)_1) \mid \\ (a \underset{snc}{:=} s)_1 \end{array} \right) \\ \parallel_{\hat{M}} s_1 \underset{snc}{\rightarrow} P_1 \parallel_{\hat{M}} b_2 \underset{snc}{\rightarrow} R \parallel_{\hat{M}} s_2 \underset{snc}{\rightarrow} P_2 \parallel_{\hat{M}} b_3 \underset{snc}{\rightarrow} P_3 \end{array} \right) \text{ in:}
$$

RHS

$\sqsupseteq$ {Definition 5.1.17 and laws 3.3.100, 3.3.40}

$$
\begin{array}{l}
(\neg chk(g))^{\top}_{S}; (chk(g_1))^{\top}_{S}; \texttt{var}\ a; \\
\quad \textbf{case} \left(\begin{array}{l} req(g); chk(g) \ ? \ \left(act(g) \parallel_{\hat{M}} (a \underset{snc}{:=} s_1)_1\right) \mid \\ req(g_1); chk(g_1) \ ? \ \left(act(g_1) \parallel_{\hat{M}} (a \underset{snc}{:=} s_2)_1\right) \mid \\ (a \underset{snc}{:=} s)_1 \end{array} \right); *P; (a = f)^{\perp}_{S}
\end{array}
$$

= {Laws 3.3.46, 4.2.40 and 4.2.39}

$$
(\neg chk(g))^{\top}_{S}; (chk(g_1))^{\top}_{S}; \texttt{var}\ a; req(g); req(g_1); (act(g_1) \parallel (a \underset{snc}{:=} s_2)_1); *P; (a = f)^{\perp}_{S}
$$

$\sqsupseteq$ {Laws 3.3.94, 4.2.6, 3.3.73, then definition 5.1.17}

$$
(\neg chk(g))^{\top}_{S}; (chk(g_1))^{\top}_{S}; req(g); req(g_1); act(g_1); a : [s_2, (P), f]
$$

$\sqsupseteq$ {Lemma 5.1.22}

$$
(\neg chk(g))^{\top}_{S}; (chk(g_1))^{\top}_{S}; req(g); req(g_1); act(g_1); a : [s_2, \left(s_2 \underset{snc}{\rightarrow} P_2 \parallel_{\hat{M}} b_3 \underset{snc}{\rightarrow} P_3\right), f]
$$

□

Our final lemma captures the case where none of the alternatives can be activated and the **priAlt** construct delays for a whole clock cycle before trying all its alternatives again.

Lemma 6.1.12.

$$
\begin{array}{l}
(\neg chk(g))^{\top}_{S}; (\neg chk(g_1))^{\top}_{S}; req(g); req(g_1); I\!I_1; \\
\quad a : [s, \left(\begin{array}{l} s \overrightarrow{}_{snc} \textbf{\textit{case}} \left(\begin{array}{l} req(g); chk(g)\ ?\ \left(act(g) \parallel_{\hat{M}} (a :=_{snc} s_1)_1 \right) \mid \\ req(g_1); chk(g_1)\ ?\ \left(act(g_1) \parallel_{\hat{M}} (a :=_{snc} s_2)_1 \right) \mid \\ (a :=_{snc} s)_1 \end{array} \right) \\ \parallel_{\hat{M}} s_1 \overrightarrow{}_{snc} P_1 \parallel_{\hat{M}} b_2 \overrightarrow{}_{snc} R \parallel_{\hat{M}} s_2 \overrightarrow{}_{snc} P_2 \parallel_{\hat{M}} b_3 \overrightarrow{}_{snc} P_3 \end{array} \right), f] \sqsubseteq \\
(\neg chk(g))^{\top}_{S}; (\neg chk(g_1))^{\top}_{S}; \\
\quad a : [s, \left(\begin{array}{l} s \overrightarrow{}_{snc} \textbf{\textit{case}} \left(\begin{array}{l} req(g); chk(g)\ ?\ \left(act(g) \parallel_{\hat{M}} (a :=_{snc} s_1)_1 \right) \mid \\ req(g_1); chk(g_1)\ ?\ \left(act(g_1) \parallel_{\hat{M}} (a :=_{snc} s_2)_1 \right) \mid \\ (a :=_{snc} s)_1 \end{array} \right) \\ \parallel_{\hat{M}} s_1 \overrightarrow{}_{snc} P_1 \parallel_{\hat{M}} b_2 \overrightarrow{}_{snc} R \parallel_{\hat{M}} s_2 \overrightarrow{}_{snc} P_2 \parallel_{\hat{M}} b_3 \overrightarrow{}_{snc} P_3 \end{array} \right), f]
\end{array}
$$

Proof.

$$
\text{Let } P = \left(\begin{array}{l} s \overrightarrow{}_{snc} \textbf{case} \left(\begin{array}{l} req(g); chk(g)\ ?\ \left(act(g) \parallel_{\hat{M}} (a :=_{snc} s_1)_1 \right) \mid \\ req(g_1); chk(g_1)\ ?\ \left(act(g_1) \parallel_{\hat{M}} (a :=_{snc} s_2)_1 \right) \mid \\ (a :=_{snc} s)_1 \end{array} \right) \\ \parallel_{\hat{M}} s_1 \overrightarrow{}_{snc} P_1 \parallel_{\hat{M}} b_2 \overrightarrow{}_{snc} R \parallel_{\hat{M}} s_2 \overrightarrow{}_{snc} P_2 \parallel_{\hat{M}} b_3 \overrightarrow{}_{snc} P_3 \end{array} \right) \text{ in:}
$$

$$
\begin{array}{l}
\quad RHS \\
\sqsupseteq \{\text{Definition 5.1.17 and laws 3.3.100, 3.3.40}\} \\
\quad (\neg chk(g))^{\top}_{S}; (\neg chk(g_1))^{\top}_{S}; \texttt{var}\ a; \\
\quad \textbf{case} \left(\begin{array}{l} req(g); chk(g)\ ?\ (act(g) \parallel (a :=_{snc} s_1)_1) \mid \\ req(g_1); chk(g_1)\ ?\ (act(g_1) \parallel (a :=_{snc} s_2)_1) \mid \\ (a :=_{snc} s)_1 \end{array} \right); *P; (a = f)^{\perp}_{S} \\
= \{\text{Laws 3.3.92, 3.3.46, 4.2.40 and 4.2.38}\} \\
\quad (\neg chk(g))^{\top}_{S}; (\neg chk(g_1))^{\top}_{S}; \texttt{var}\ a; req(g); req(g_1); (a :=_{snc} s)_1; *P; (a = f)^{\perp}_{S} \\
\sqsupseteq \{\text{Laws 3.3.94, 4.2.6, 3.3.73, then definition 5.1.17}\} \\
\quad (\neg chk(g))^{\top}_{S}; (\neg chk(g_1))^{\top}_{S}; req(g); req(g_1); I\!I_1; a : [s, (P), f] \\
= \{\text{Definition of } P\} \\
\quad LHS
\end{array}
$$

□

The compilation theorem relating the non-default **priAlt** construct to its corresponding normal form can now be proved using the results above.

Theorem 6.1.13. *Non-default* ***priAlt*** *first normal form*

$$\textbf{\textit{priAlt}}\left\{\begin{array}{l}\textbf{\textit{case}}\ g_1\textbf{\textit{:}}\ a:[s_1,\left(s_1 \underset{snc}{\rightarrow} P_1 \parallel_{\hat{M}} b_2 \underset{snc}{\rightarrow} R\right),f]\ ;\ \textbf{\textit{break}};\\ \textbf{\textit{case}}\ g_2\textbf{\textit{:}}\ a:[s_2,\left(s_2 \underset{snc}{\rightarrow} P_2 \parallel_{\hat{M}} b_3 \underset{snc}{\rightarrow} P_3\right),f]\ ;\ \textbf{\textit{break}}\end{array}\right\} \sqsubseteq$$

$$a:[s,\left(\begin{array}{l} s \underset{snc}{\rightarrow} \textbf{\textit{case}}\left(\begin{array}{l} req(g);chk(g)\ ?\ (act(g) \parallel (a \underset{snc}{:=} s_1)_1)\ |\\ req(g_1);chk(g_1)\ ?\ (act(g_1) \parallel (a \underset{snc}{:=} s_2)_1)\ |\\ (a \underset{snc}{:=} s)_1\end{array}\right)\\ \parallel_{\hat{M}} s_1 \underset{snc}{\rightarrow} P_1 \parallel_{\hat{M}} b_2 \underset{snc}{\rightarrow} R \parallel_{\hat{M}} s_2 \underset{snc}{\rightarrow} P_2 \parallel_{\hat{M}} b_3 \underset{snc}{\rightarrow} P_3\end{array}\right),f]$$

Proof.

$$\begin{array}{l} RHS = RHS\\ \equiv \{\text{Laws 3.3.23(twice) and 3.3.57}\}\\ RHS = \left(chk(g)^{\top};RHS\right) \lhd chk(g) \rhd\\ \qquad \left(\left((\neg chk(g))^{\top}_{S};chk(g_1)^{\top};RHS\right) \lhd chk(g_1) \rhd \left((\neg chk(g))^{\top}_{S};(\neg chk(g_1))^{\top}_{S};RHS\right)\right)\\ \equiv \{\text{Lemmas 6.1.10, 6.1.11, 6.1.12, then law 3.3.57}\}\\ RHS \sqsupseteq \left(req(g);act(g);a:[s_1,\left(s_1 \underset{snc}{\rightarrow} P_1 \parallel_{\hat{M}} b_2 \underset{snc}{\rightarrow} R\right),f]\right) \lhd chk(g) \rhd\\ \qquad (\left(req(g);req(g_1);act(g_1);a:[s_1,\left(s_2 \underset{snc}{\rightarrow} P_2 \parallel_{\hat{M}} b_3 \underset{snc}{\rightarrow} P_3\right),f]\right) \lhd chk(g_1) \rhd\\ \qquad (req(g);req(g_1);I\!I_1;RHS))\\ \Rightarrow \{\text{Laws 2.3.81 and 4.2.44}\}\\ RHS \sqsupseteq \mu X \bullet req(g);\left(act(g);a:[s_1,\left(s_1 \underset{snc}{\rightarrow} P_1 \parallel_{\hat{M}} b_2 \underset{snc}{\rightarrow} R\right),f]\right) \lhd chk(g) \rhd\\ \qquad req(g_1);(\left(act(g_1);a:[s_1,\left(s_2 \underset{snc}{\rightarrow} P_2 \parallel_{\hat{M}} b_3 \underset{snc}{\rightarrow} P_3\right),f]\right) \lhd chk(g_1) \rhd I\!I_1;X)\\ \equiv \{\text{Definition 4.2.34}\}\\ RHS \sqsupseteq \mu X \bullet \textbf{case}\left(\begin{array}{l} req(g);chk(g)\ ?\ act(g);a:[s_1,\left(s_1 \underset{snc}{\rightarrow} P_1 \parallel_{\hat{M}} b_2 \underset{snc}{\rightarrow} R\right),f]\ |\\ req(g_1);chk(g_1)\ ?\ act(g_1);a:[s_1,\left(s_2 \underset{snc}{\rightarrow} P_2 \parallel_{\hat{M}} b_3 \underset{snc}{\rightarrow} P_3\right),f]\ |\\ I\!I_1;X\end{array}\right)\\ \equiv \{\text{Theorem 4.2.47}\}\\ RHS \sqsupseteq LHS \end{array}$$

□

6.2 Second normal form: eliminating communication primitives

In Section 5.2 we have shown how to eliminate parallel by merge from a normal form where each step is a guarded multiple assignment. Clearly, this technique cannot be directly applied to the communication-based normal forms described in this chapter. The main limiting factor towards being able to reuse the compilation theorems from the previous chapter is the presence of the communication requests, conditionals and transmission primitives used in the the normal form encoding of input, output and **priAlt**.

In this section we provide simplification rules that allow us to re-formulate the channel operations and case construct used in the communication normal forms in terms of guarded assignments. The key idea behind our strategy is that a step of the form

$$b \overrightarrow{}_{\text{snc}} \mathbf{in\text{-}req}(ch); P$$

where P does not depend on the input request over ch can be expressed as the composition:

$$b \overrightarrow{}_{\text{snc}} \mathbf{in\text{-}req}(ch)_1 \parallel_{\hat{M}} b \overrightarrow{}_{\text{snc}} P$$

From this alternative formulation it is clear that a request for input over channel ch will be issued when b holds. In this context $b \Leftrightarrow \mathbf{in\text{-}req}(ch) \Leftrightarrow \mathbf{rd}(ch)$, allowing us to use b instead of $\mathbf{rd}(ch)$ in the normal form. A similar strategy can be applied with $\mathbf{out\text{-}req}(ch)$ – $\mathbf{wr}(ch)$ and $\mathbf{out}(ch, e)$ – $\mathbf{in}(ch)$ pairs. The rest of this section is devoted to describing the strategy described above in further detail.

6.2.1 Simplifying steps from input and output

The first transformation to our normal form allows us to split the actions within a step in order to expose input and output requests. In this way, we separate the control flow (requests from input and output) from the data flow (transfer of values) within communicating steps.

Law 6.2.1. *Input normal form re-structuring*

$$a : [a = s, (a = s \overrightarrow{}_{snc} (\boldsymbol{in\text{-}req}(ch); x, a \mathrel{:=}_{snc} (f, \boldsymbol{in}(ch)) \lhd \boldsymbol{wr}(ch) \rhd (s, x))_1), a = f] =$$
$$a : [a = s, \begin{pmatrix} a = s \overrightarrow{}_{snc} \boldsymbol{in\text{-}req}(ch)_1 \parallel_{\hat{M}} \\ a = s \overrightarrow{}_{snc} (x \mathrel{:=}_{snc} \boldsymbol{in}(ch) \lhd \boldsymbol{wr}(ch) \rhd x)_1 \parallel_{\hat{M}} \\ a = s \overrightarrow{}_{snc} (a \mathrel{:=}_{snc} f \lhd \boldsymbol{wr}(ch) \rhd s)_1 \end{pmatrix}, a = f]$$

Proof.

$$\begin{array}{ll} & a : [a = s, (a = s \overrightarrow{}_{\text{snc}} (\mathbf{in\text{-}req}(ch); x, a \mathrel{:=}_{\text{snc}} (f, \mathbf{in}(ch)) \lhd \mathbf{wr}(ch) \rhd (s, x))_1), a = f] \\ = & \{\text{Definition 5.1.17, law 3.3.146}\} \\ & \texttt{var}\ a; (a = s)^{\top}_{\mathsf{S}}; \\ & \quad a = s \overrightarrow{}_{\text{snc}} \left(\mathbf{in\text{-}req}(ch); ((x \mathrel{:=}_{\text{snc}} \mathbf{in}(ch) \lhd \mathbf{wr}(ch) \rhd x)_1 \parallel_{\hat{M}} (a \mathrel{:=}_{\text{snc}} f \lhd \mathbf{wr}(ch) \rhd s))_1\right); \\ & (a = f)^{\bot}_{\mathsf{S}}; \texttt{end}\ a \\ = & \{\text{Law 4.2.27}\} \\ & \texttt{var}\ a; (a = s)^{\top}_{\mathsf{S}}; \\ & \quad a = s \overrightarrow{}_{\text{snc}} \left(\mathbf{in\text{-}req}(ch)_1 \parallel_{\hat{M}} (x \mathrel{:=}_{\text{snc}} \mathbf{in}(ch) \lhd \mathbf{wr}(ch) \rhd x)_1 \parallel_{\hat{M}} (a \mathrel{:=}_{\text{snc}} f \lhd \mathbf{wr}(ch) \rhd s)_1\right); \\ & (a = f)^{\bot}_{\mathsf{S}}; \texttt{end}\ a \end{array}$$

= {Law 3.3.155, definition 5.1.17}

$$a : [a = s, \begin{pmatrix} a = s \overrightarrow{}_{\text{snc}} \textbf{in-req}(ch)_1 \parallel_{\hat{M}} \\ a = s \overrightarrow{}_{\text{snc}} (x \mathrel{:=}_{\text{snc}} \textbf{in}(ch) \lhd \textbf{wr}(ch) \rhd x)_1 \parallel_{\hat{M}} \\ a = s \overrightarrow{}_{\text{snc}} (a \mathrel{:=}_{\text{snc}} f \lhd \textbf{wr}(ch) \rhd s)_1 \end{pmatrix}, a = f] \qquad \square$$

A similar transformation can be applied to steps within output normal forms:

Law 6.2.2. *Output normal form re-structuring*

$$a : [a = s, (a = s \overrightarrow{}_{snc} \textbf{\textit{out-req}}(ch); \textbf{\textit{out}}(ch, e); (v, a \mathrel{:=}_{snc} v, (f \lhd \textbf{\textit{rd}}(ch) \rhd s))_1), a = f] =$$

$$a : [a = s, \begin{pmatrix} a = s \overrightarrow{}_{snc} \textbf{\textit{out-req}}(ch)_1 \parallel_{\hat{M}} \\ a = s \overrightarrow{}_{snc} \textbf{\textit{out}}(ch, e)_1 \parallel_{\hat{M}} \\ a = s \overrightarrow{}_{snc} a \mathrel{:=}_{snc} (f \lhd \textbf{\textit{rd}}(ch) \rhd s)_1 \end{pmatrix}, a = f]$$

Proof.

Similar to the proof of law 6.2.1. $\square$

Assuming the whole program uses only one channel and, two inputs from that channel (to variables x and y respectively)[1], we can apply the transformations described in Laws 6.2.1 and 6.2.2, to get our program in the following form:

$$a : [s, \begin{pmatrix} s_1 \overrightarrow{}_{\text{snc}} \textbf{in-req}(ch)_1 \parallel_{\hat{M}} s_2 \overrightarrow{}_{\text{snc}} \textbf{in-req}(ch)_1 \parallel_{\hat{M}} \\ s_3 \overrightarrow{}_{\text{snc}} a \mathrel{:=}_{\text{snc}} (s_3 \lhd \textbf{rd}(ch) \rhd f_3)_1 \parallel_{\hat{M}} P \end{pmatrix}, f]$$

where P does not perform any **in-req**(ch). Notice we have highlighted the parts of the normal form where the input request is issued (two of them, because we are performing two inputs) and the portion of the normal form that depends on a reader to perform output into the channel. The guarded commands performing the remaining actions for the input and output commands are 'hidden' within P.

Our intention is now to eliminate all occurrences of **in-req**(ch) and **rd**(ch) from the normal form. Notice that the first line is indicating (by means of a guarded command) under which control states the request for input will be issued. If we consider that **rd**(ch) will only hold true provided at least a process is willing to perform an input into the channel, we should be able to replace the condition **rd**(ch) by the conditions guarding the issue of the input request. The following theorem captures this notion.

[1]Both assumptions (a single channel in the program and a two input commands) are in place only to keep the presentation compact, the same approach will be valid for a larger number of channels (by means of considering them one at a time) and input/output commands.

Lemma 6.2.3. *Input request and condition elimination*

$$a : [s, \begin{pmatrix} P \parallel_{\hat{M}} \\ s_1 \overrightarrow{}_{snc} \textbf{\textit{in-req}}(ch)_1 \parallel_{\hat{M}} \\ s_2 \overrightarrow{}_{snc} \textbf{\textit{in-req}}(ch)_1 \parallel_{\hat{M}} \\ s_3 \overrightarrow{}_{snc} a :=_{snc} (j \lhd \textbf{\textit{rd}}(ch) \rhd s_3)_1 \end{pmatrix}, f] = a : [s, \begin{pmatrix} P[s_1 \vee s_2/\textbf{\textit{rd}}(ch)] \\ \parallel_{\hat{M}} s_3 \overrightarrow{}_{snc} a :=_{snc} (j \lhd (s_1 \vee s_2) \rhd s_3)_1 \end{pmatrix}, f]$$

Proof.

$$\begin{aligned}
& a : [s, \begin{pmatrix} s_1 \overrightarrow{}_{snc} \textbf{in-req}(ch)_1 \parallel_{\hat{M}} s_2 \overrightarrow{}_{snc} \textbf{in-req}(ch)_1 \parallel_{\hat{M}} \\ s_3 \overrightarrow{}_{snc} a :=_{snc} (j \lhd \textbf{rd}(ch) \rhd s_3)_1 \parallel_{\hat{M}} P \end{pmatrix}, f] \\
= & \{\text{Law 3.3.156}\} \\
& a : [s, \Big((s_1 \vee s_2) \overrightarrow{}_{snc} \textbf{in-req}(ch)_1 \parallel_{\hat{M}} s_3 \overrightarrow{}_{snc} a :=_{snc} (s_3 \lhd \textbf{rd}(ch) \rhd f_3)_1 \parallel_{\hat{M}} P\Big), f] \\
= & \{\text{Law 4.2.29}\} \\
& a : [s, \Big((s_3 \overrightarrow{}_{snc} a :=_{snc} (j \lhd \textbf{rd}(ch) \rhd s_3)_1 \parallel_{\hat{M}} P)\,[(s_1 \vee s_2)/\textbf{rd}(ch)]\Big), f] \\
= & \{\text{Propositional calculus}\} \\
& a : [s, \Big(s_3 \overrightarrow{}_{snc} a :=_{snc} (j \lhd (s_1 \vee s_2) \rhd s_3)_1 \parallel_{\hat{M}} P\Big), f]
\end{aligned}$$

□

We can a apply a similar approach to eliminate all occurrences of **out-req**(ch), and **wr**(ch) from the normal form.

Lemma 6.2.4. *Output request and condition elimination*

$$a : [s, \begin{pmatrix} P \parallel_{\hat{M}} \\ s_1 \overrightarrow{}_{snc} \textbf{\textit{out-req}}(ch)_1 \parallel_{\hat{M}} \\ s_2 \overrightarrow{}_{snc} \textbf{\textit{out-req}}(ch)_1 \parallel_{\hat{M}} \\ s_3 \overrightarrow{}_{snc} (a :=_{snc} j \lhd \textbf{\textit{wr}}(ch) \rhd s_3)_1 \parallel_{\hat{M}} \\ s_3 \overrightarrow{}_{snc} (x :=_{snc} \textbf{\textit{in}}(ch) \lhd \textbf{\textit{wr}}(ch) \rhd x)_1 \end{pmatrix}, f] = a : [s, \begin{pmatrix} P[s_1 \vee s_2/\textbf{\textit{wr}}(ch)] \parallel_{\hat{M}} \\ s_3 \overrightarrow{}_{snc} a :=_{snc} (j \lhd b \rhd s_3)_1 \parallel_{\hat{M}} \\ s_3 \overrightarrow{}_{snc} (x :=_{snc} \textbf{\textit{in}}(ch) \lhd b \rhd x)_1 \end{pmatrix}, f]$$

where $b = (s_1 \vee s_2)$.

Proof.

Similar to proof of law 6.2.3, but using law 4.2.30. □

To complete the simplification of the input and output commands we need to devise a way of dealing with the functions actually writing to and reading from a channel ch (i.e., **out**(ch, e) and **in**(ch) respectively). The idea for this simplification is that processes outputting to the channel will only effectively transmit data when their guard is activated. As more than one writer has access to the channel (even though they should not write during the same clock cycle), the value being transmitted over the channel will have to be selected from the appropriate source. This selected value can be then directly fed to the variable receiving data from the channel. These ideas are more formally captured by our final simplification theorem for channels.

Lemma 6.2.5. *Channel transmission primitives elimination*

$$a : [s, \begin{pmatrix} s_1 \overrightarrow{}_{snc} \textbf{\textit{out}}(ch, e_1)_1 \parallel_{\hat{M}} \\ s_2 \overrightarrow{}_{snc} \textbf{\textit{out}}(ch, e_2)_1 \parallel_{\hat{M}} \\ s_3 \overrightarrow{}_{snc} (x :=_{snc} \textbf{\textit{in}}(ch) \lhd d \rhd x)_1 \parallel_{\hat{M}} \\ s_4 \overrightarrow{}_{snc} (y :=_{snc} \textbf{\textit{in}}(ch) \lhd d \rhd y)_1 \parallel_{\hat{M}} \\ P \end{pmatrix}, f] \sqsubseteq$$

$$a : [s, \begin{pmatrix} s_3 \overrightarrow{}_{snc} (x :=_{snc} (\textit{SELECT}(\textit{ARB}, (e_1 \lhd s_1 \rhd \textit{ARB}), (e_2 \lhd s_2 \rhd \textit{ARB}))) \lhd d \rhd x)_1 \parallel_{\hat{M}} \\ s_4 \overrightarrow{}_{snc} (y :=_{snc} (\textit{SELECT}(\textit{ARB}, (e_1 \lhd s_1 \rhd \textit{ARB}), (e_2 \lhd s_2 \rhd \textit{ARB}))) \lhd d \rhd y)_1 \parallel_{\hat{M}} \\ P \end{pmatrix}, f]$$

where $d = (s_1 \vee s_2)$ and $v = (e_1 \lhd s_1 \rhd e_2)$.

Proof.

$$a : [s, \begin{pmatrix} s_1 \overrightarrow{}_{\text{snc}} \textbf{out}(ch, e_1)_1 \parallel_{\hat{M}} \\ s_2 \overrightarrow{}_{\text{snc}} \textbf{out}(ch, e_2)_1 \parallel_{\hat{M}} \\ s_3 \overrightarrow{}_{\text{snc}} (x :=_{\text{snc}} \textbf{in}(ch) \lhd d \rhd x)_1 \parallel_{\hat{M}} \\ s_4 \overrightarrow{}_{\text{snc}} (y :=_{\text{snc}} \textbf{in}(ch) \lhd d \rhd y)_1 \parallel_{\hat{M}} \\ P \end{pmatrix}, f]$$

$\sqsubseteq$ {Law 4.2.31, propositional calculus}

$$a : [s, \begin{pmatrix} s_3 \overrightarrow{}_{\text{snc}} (x :=_{\text{snc}} (\texttt{SELECT}\,(\texttt{ARB}, (e_1 \lhd s_1 \rhd \texttt{ARB}), (e_2 \lhd s_2 \rhd \texttt{ARB}))) \lhd d \rhd x)_1 \parallel_{\hat{M}} \\ s_4 \overrightarrow{}_{\text{snc}} (y :=_{\text{snc}} (\texttt{SELECT}\,(\texttt{ARB}, (e_1 \lhd s_1 \rhd \texttt{ARB}), (e_2 \lhd s_2 \rhd \texttt{ARB}))) \lhd d \rhd y)_1 \parallel_{\hat{M}} \\ P \end{pmatrix}, f] \; \square$$

After applying theorems 6.2.3, 6.2.4 and 6.2.5, the steps produced by input or output commands are transformed into one-clock cycle assignments. From this point, the results from Section 5.2.2 can be applied to reduce these commands to second normal form.

6.2.2 Simplifying steps from prioritised choice

Based on the way in which we have reduced prioritised choice to first normal form, it is evident that all non-assignment constructs will only arise within the first step in the normal form associated to the **priAlt** construct. The techniques from the previous section will allow us to eliminate the communication primitives so we only need to find a way of eliminating the **case** expression from the right hand side of the guarded command in the first step.

The foundation for our strategy for **case** expressions elimination is that they are, in essence, a combinatorial action (in particular, an input or output request) followed by a conditional. In this context we only need to find a way of simplifying guarded commands where the guarded expression is a condition. It turns out that this is not a problem as we can express constructs of the

form

$$b_1 \overrightarrow{;}_{\text{snc}} (P \lhd b_2 \rhd Q)$$

in the parallel form

$$(b_1 \wedge b_2) \overrightarrow{;}_{\text{snc}} P \parallel_{\hat{M}} (b_1 \wedge \neg b_2) \overrightarrow{;}_{\text{snc}} Q$$

(law 3.3.157). The following lemma applies this notion to the context of case statements:

Lemma 6.2.6. *Case statement elimination*

$$s \overrightarrow{;}_{snc} \textbf{\textit{case}} \left(\begin{array}{l} req(g); chk(g) \ ? \ \left(act(g)_1 \parallel_{\hat{M}} (a :=_{\text{snc}} s_1)_1 \right) \mid \\ P \end{array} \right) = \left(\begin{array}{l} s \overrightarrow{;}_{snc} req(g)_1 \parallel_{\hat{M}} \\ (s \wedge chk(g)) \overrightarrow{;}_{snc} act(g)_1 \parallel_{\hat{M}} \\ (s \wedge chk(g)) \overrightarrow{;}_{snc} (a :=_{snc} s_1)_1 \parallel_{\hat{M}} \\ (s \wedge \neg chk(g)) \overrightarrow{;}_{snc} P \end{array} \right)$$

Proof.

$$\begin{array}{l} \quad s \overrightarrow{;}_{\text{snc}} \textbf{case} \ \left(req(g); chk(g) \ ? \ \left(act(g)_1 \parallel_{\hat{M}} (a :=_{\text{snc}} s_1)_1 \right) \mid P \right) \\ = \{\text{Definition 4.2.34}\} \\ \quad s \overrightarrow{;}_{\text{snc}} req(g); \left(\left(act(g)_1 \parallel_{\hat{M}} (a :=_{\text{snc}} s_1)_1 \right) \lhd chk(g) \rhd P \right) \\ = \{\text{Laws 4.2.27 and 4.2.28}\} \\ \quad s \overrightarrow{;}_{\text{snc}} req(g)_1 \parallel_{\hat{M}} s \overrightarrow{;}_{\text{snc}} \left(\left(act(g)_1 \parallel_{\hat{M}} (a :=_{\text{snc}} s_1)_1 \right) \lhd chk(g) \rhd P \right) \\ = \{\text{Law 3.3.157}\} \\ \quad s \overrightarrow{;}_{\text{snc}} req(g)_1 \parallel_{\hat{M}} (s \wedge chk(g)) \overrightarrow{;}_{\text{snc}} \left(act(g)_1 \parallel_{\hat{M}} (a :=_{\text{snc}} s_1)_1 \right) \parallel_{\hat{M}} (s \wedge \neg chk(g)) \overrightarrow{;}_{\text{snc}} P \\ = \{\text{Law 3.3.155}\} \\ \quad s \overrightarrow{;}_{\text{snc}} req(g)_1 \parallel_{\hat{M}} (s \wedge chk(g)) \overrightarrow{;}_{\text{snc}} act(g)_1 \parallel_{\hat{M}} (s \wedge chk(g)) \overrightarrow{;}_{\text{snc}} (a :=_{\text{snc}} s_1)_1 \parallel_{\hat{M}} (s \wedge \neg chk(g)) \overrightarrow{;}_{\text{snc}} P \ \square \end{array}$$

Note that, in case the whole step was a nested case statement (i.e., P is a **case** expression), then the last step in our simplified normal form will have the form

$$(s \wedge \neg chk(g)) \overrightarrow{;}_{\text{snc}} \textbf{case} \ (req(g); chk(g) \ ? \ A_1 \mid A_2)$$

this will allow us to keep applying Lemma 6.2.6 until the **case** expression is completely eliminated.

6.2.3 Dealing with impossible communications

So far we have provided a way of dealing with input, output and prioritised choice that relies on the fact that every program contains at least one **in-req**(ch) to match up with the occurrences of the **rd**(ch) construct in the program (and, similarly, an **out-req**(ch) to match the condition **wr**(ch)). It is possible, however, to write a program that has an input/output command without its counterpart.

In this case, the moment the control flow reaches the input/output command, the program will iterate forever (trying to communicate, failing, delaying one clock cycle and trying again).

In our compiler, the first normal form version of the same program will also diverge. However, when trying to transform the program into second normal form, the laws from Section 6.2.1 and 6.2.2 will not be enough to eliminate the communicating conditions associated with this unmatched input/output commands. More formally, the compilation will not be able to proceed any further after the reduction reaches a state of the form:

$$a : [s, \left(s_1 \underset{snc}{\rightarrow} (y \underset{snc}{:=} v_1 \lhd chk(ch) \rhd v_2)_1 \parallel_{\hat{M}} P\right), f]$$

where P is the parallel by merge combination of guarded multiple assignments and it does not engage in any $\mathbf{req}(ch)$ event[2]. We can take advantage of the knowledge that the $chk(ch)$ condition will never hold in order to simplify the program. The following lemma states this fact more accurately.

Lemma 6.2.7. *Impossible communication simplification*

$$a : [s, \left(s_1 \underset{snc}{\rightarrow} (y \underset{snc}{:=} v_1 \lhd chk(ch) \rhd v_2)_1 \parallel_{\hat{M}} P\right), f] = a : [s, \left(s_1 \underset{snc}{\rightarrow} (y \underset{snc}{:=} v_2)_1 \parallel_{\hat{M}} P[\mathbf{false}/chk(ch)]\right), f]$$

provided that P does not engage in any $\mathbf{req}(ch)$ event and that the normal form encompasses the whole program.

Proof.

$$\begin{aligned}
& a : [s, \left(s_1 \underset{snc}{\rightarrow} (y \underset{snc}{:=} v_1 \lhd chk(ch) \rhd v_2)_1 \parallel_{\hat{M}} P\right), f] \\
= & \{\text{Definitions 3.3.149 and 3.2.7, law 3.3.28}\} \\
& a : [s, \left((y \underset{snc}{:=} (v_1 \lhd chk(ch) \rhd v_2) \lhd s_1 \rhd y)_1 \parallel_{\hat{M}} P\right), f] \\
= & \{\text{Assumption } (P \text{ does not engage in any } \mathbf{req}(ch) \text{ event}), \text{laws 4.2.32 and 4.2.33}\} \\
& a : [s, \left(((y \underset{snc}{:=} (v_1 \lhd chk(ch) \rhd v_2) \lhd s_1 \rhd y)_1 \parallel_{\hat{M}} P)[\mathbf{false}/chk(ch)]\right), f] \\
= & \{\text{Predicate calculus, law 3.3.25}\} \\
& a : [s, \left((y \underset{snc}{:=} v_2 \lhd s_1 \rhd y)_1 \parallel_{\hat{M}} P[\mathbf{false}/chk(ch)]\right), f] \\
= & \{\text{Definitions 3.3.149 and 3.2.7, law 3.3.28}\} \\
& a : [s, \left(s_1 \underset{snc}{\rightarrow} (y \underset{snc}{:=} v_2)_1 \parallel_{\hat{M}} P[\mathbf{false}/chk(ch)]\right), f]
\end{aligned}$$

□

6.3 Putting it all together

In this section we briefly illustrate usage of the compilation theorems outlined in this section for the reduction of the communication primitives and the **priAlt** construct into second normal form.

[2]Note here that there might be more than one step depending on $chk(ch)$ that cannot be resolved. The case above is presented with illustrative purposes only, and it is kept simple with this idea in mind. The technique described to deal with this case will as well apply for the case with more than one step depending on $chk(ch)$

The example we will compile is the parallel composition of an input command with a **priAlt** construct containing an input and an output command:

$$ch?x \underset{\mathsf{HC}}{\parallel} \mathbf{priAlt} \left\{ \begin{array}{l} \mathbf{case}\ ch_1?x\mathbf{:}\ \mathbf{delay} \fatsemi \mathbf{break} \fatsemi \\ \mathbf{case}\ ch!0\mathbf{:}\ \mathbf{delay} \fatsemi \mathbf{break} \end{array} \right\}$$

We begin by reducing the input and the **priAlt** commands into normal form:

$$\begin{array}{l} a_l : [a_l = s, (a_l = s \overset{\rightarrow}{}_{\mathrm{snc}} (\mathbf{in\text{-}req}(ch); a_l, x \overset{:=}{}_{\mathrm{snc}} (f, \mathbf{in}(ch)) \lhd \mathbf{wr}(ch) \rhd (s, x))_1), a_l = f] \\ \underset{\mathsf{HC}}{\parallel} \\ a_r : [a_r = s, \left(\begin{array}{l} a_r = s \overset{\rightarrow}{}_{\mathrm{snc}} \mathbf{case} \left(\begin{array}{l} \mathbf{in\text{-}req}(ch_1); \mathbf{wr}(ch_1)\ ?\ (x, a_r \overset{:=}{}_{\mathrm{snc}} \mathbf{in}(ch_1), s_1)_1 \mid \\ \mathbf{out\text{-}req}(ch); \mathbf{out}(ch, 0); \mathbf{rd}(ch)\ ?\ (a_r \overset{:=}{}_{\mathrm{snc}} s_2)_1 \mid \\ (a_r \overset{:=}{}_{\mathrm{snc}} s)_1 \end{array} \right) \\ \|_{\hat{M}}\ a_r = s_1 \overset{\rightarrow}{}_{\mathrm{snc}} (a_r \overset{:=}{}_{\mathrm{snc}} f)_1 \|_{\hat{M}}\ a_r = s_2 \overset{\rightarrow}{}_{\mathrm{snc}} (a_r \overset{:=}{}_{\mathrm{snc}} f)_1 \end{array} \right), a_r = f] \end{array}$$

Reducing the parallel composition operator into normal form and making

$$b_1 = (a_l = s_0 \wedge a_r = s_0)$$

and

$$b_1 = (a_l = f \wedge a_r = f)$$

we obtain:

$$a_l, a_r : [b_1, \left(\begin{array}{l} b_1 \overset{\rightarrow}{}_{\mathrm{snc}} (\mathbf{in\text{-}req}(ch); a_l, x \overset{:=}{}_{\mathrm{snc}} (f, \mathbf{in}(ch)) \lhd \mathbf{wr}(ch) \rhd (s, x))_1 \|_{\hat{M}} \\ b_1 \overset{\rightarrow}{}_{\mathrm{snc}} \mathbf{case} \left(\begin{array}{l} \mathbf{in\text{-}req}(ch_1); \mathbf{wr}(ch_1)\ ?\ (x, a_r \overset{:=}{}_{\mathrm{snc}} \mathbf{in}(ch_1), s_1)_1 \mid \\ \mathbf{out\text{-}req}(ch); \mathbf{out}(ch, 0); \mathbf{rd}(ch)\ ?\ (a_r \overset{:=}{}_{\mathrm{snc}} s_2)_1 \mid \\ (a_r \overset{:=}{}_{\mathrm{snc}} s)_1 \end{array} \right) \\ a_l = s \overset{\rightarrow}{}_{\mathrm{snc}} (\mathbf{in\text{-}req}(ch); a_l, x \overset{:=}{}_{\mathrm{snc}} (f, \mathbf{in}(ch)) \lhd \mathbf{wr}(ch) \rhd (s, x))_1 \|_{\hat{M}} \\ a_r = s \overset{\rightarrow}{}_{\mathrm{snc}} \mathbf{case} \left(\begin{array}{l} \mathbf{in\text{-}req}(ch_1); \mathbf{wr}(ch_1)\ ?\ (x, a_r \overset{:=}{}_{\mathrm{snc}} \mathbf{in}(ch_1), s_1)_1 \mid \\ \mathbf{out\text{-}req}(ch); \mathbf{out}(ch, 0); \mathbf{rd}(ch)\ ?\ (a_r \overset{:=}{}_{\mathrm{snc}} s_2)_1 \mid \\ (a_r \overset{:=}{}_{\mathrm{snc}} s)_1 \end{array} \right) \\ \|_{\hat{M}}\ a_r = s_1 \overset{\rightarrow}{}_{\mathrm{snc}} (a_r \overset{:=}{}_{\mathrm{snc}} f)_1 \|_{\hat{M}}\ a_r = s_2 \overset{\rightarrow}{}_{\mathrm{snc}} (a_r \overset{:=}{}_{\mathrm{snc}} f)_1 \end{array} \right), b_2]$$

Applying law 6.2.1 and Lemma 6.2.6 (twice) we get:

$$a_l, a_r : [b_1, \left(\begin{array}{l} b_1 \overrightarrow{}_{\text{snc}} \textbf{in-req}(ch)_1 \parallel_{\hat{M}} b_1 \overrightarrow{}_{\text{snc}} (a_l, x \mathrel{:=}_{\text{snc}} (f, \textbf{in}(ch)) \lhd \textbf{wr}(ch) \rhd (s, x))_1 \parallel_{\hat{M}} \\ b_1 \overrightarrow{}_{\text{snc}} \textbf{in-req}(ch_1)_1 \parallel_{\hat{M}} \\ b_1 \wedge \textbf{wr}(ch_1) \overrightarrow{}_{\text{snc}} (x, a_r \mathrel{:=}_{\text{snc}} \textbf{in}(ch_1), s_1)_1 \parallel_{\hat{M}} \\ b_1 \wedge \neg \textbf{wr}(ch_1) \overrightarrow{}_{\text{snc}} \textbf{out-req}(ch)_1 \parallel_{\hat{M}} \\ b_1 \wedge \neg \textbf{wr}(ch_1) \overrightarrow{}_{\text{snc}} \textbf{out}(ch, 0)_1 \parallel_{\hat{M}} \\ b_1 \wedge \neg \textbf{wr}(ch_1) \wedge \textbf{rd}(ch) \overrightarrow{}_{\text{snc}} (a_r \mathrel{:=}_{\text{snc}} s_2)_1 \parallel_{\hat{M}} \\ b_1 \wedge \neg \textbf{wr}(ch_1) \wedge \neg \textbf{rd}(ch) \overrightarrow{}_{\text{snc}} (a_r \mathrel{:=}_{\text{snc}} s)_1 \parallel_{\hat{M}} \\ a_l = s \overrightarrow{}_{\text{snc}} \textbf{in-req}(ch)_1 \parallel_{\hat{M}} a_r = s \overrightarrow{}_{\text{snc}} \textbf{in-req}(ch_1)_1 \parallel_{\hat{M}} \\ a_r = s \overrightarrow{}_{\text{snc}} (a_l, x \mathrel{:=}_{\text{snc}} (f, \textbf{in}(ch)) \lhd \textbf{wr}(ch) \rhd (s, x))_1 \parallel_{\hat{M}} \\ a_r = s \wedge \textbf{wr}(ch_1) \overrightarrow{}_{\text{snc}} (x, a_r \mathrel{:=}_{\text{snc}} \textbf{in}(ch_1), s_1)_1 \parallel_{\hat{M}} \\ a_r = s \wedge \neg \textbf{wr}(ch_1) \overrightarrow{}_{\text{snc}} \textbf{out-req}(ch)_1 \parallel_{\hat{M}} \\ a_r = s \wedge \neg \textbf{wr}(ch_1) \overrightarrow{}_{\text{snc}} \textbf{out}(ch, 0)_1 \parallel_{\hat{M}} \\ a_r = s \wedge \neg \textbf{wr}(ch_1) \wedge \textbf{rd}(ch) \overrightarrow{}_{\text{snc}} (a_r \mathrel{:=}_{\text{snc}} s_2)_1 \parallel_{\hat{M}} \\ a_r = s \wedge \neg \textbf{wr}(ch_1) \wedge \neg \textbf{rd}(ch) \overrightarrow{}_{\text{snc}} (a_r \mathrel{:=}_{\text{snc}} s)_1 \parallel_{\hat{M}} \\ a_r = s_1 \overrightarrow{}_{\text{snc}} (a_r \mathrel{:=}_{\text{snc}} f)_1 \parallel_{\hat{M}} a_r = s_2 \overrightarrow{}_{\text{snc}} (a_r \mathrel{:=}_{\text{snc}} f)_1 \end{array}\right), b_2]$$

If we now apply Lemmas 6.2.3 and 6.2.4 , letting $b_3 = (b_1 \wedge \neg \textbf{wr}(ch_1)) \vee (a_r = s \wedge \neg \textbf{wr}(ch_1))$ we obtain:

$$a_l, a_r : [b_1, \left(\begin{array}{l} b_1 \overrightarrow{}_{\text{snc}} (a_l, x \mathrel{:=}_{\text{snc}} (f, \textbf{in}(ch)) \lhd b_3 \rhd (s, x))_1 \parallel_{\hat{M}} \\ b_1 \wedge \textbf{wr}(ch_1) \overrightarrow{}_{\text{snc}} (x, a_r \mathrel{:=}_{\text{snc}} \textbf{in}(ch_1), s_1)_1 \parallel_{\hat{M}} \\ b_1 \wedge \neg \textbf{wr}(ch_1) \overrightarrow{}_{\text{snc}} \textbf{out}(ch, 0)_1 \parallel_{\hat{M}} \\ b_1 \wedge \neg \textbf{wr}(ch_1) \wedge (b_1 \vee a_l = s) \overrightarrow{}_{\text{snc}} (a_r \mathrel{:=}_{\text{snc}} s_2)_1 \parallel_{\hat{M}} \\ b_1 \wedge \neg \textbf{wr}(ch_1) \wedge \neg (b_1 \vee a_l = s) \overrightarrow{}_{\text{snc}} (a_r \mathrel{:=}_{\text{snc}} s)_1 \parallel_{\hat{M}} \\ a_r = s \overrightarrow{}_{\text{snc}} (a_l, x \mathrel{:=}_{\text{snc}} (f, \textbf{in}(ch)) \lhd b_3 \rhd (s, x))_1 \parallel_{\hat{M}} \\ a_r = s \wedge \textbf{wr}(ch_1) \overrightarrow{}_{\text{snc}} (x, a_r \mathrel{:=}_{\text{snc}} \textbf{in}(ch_1), s_1)_1 \parallel_{\hat{M}} \\ a_r = s \wedge \neg \textbf{wr}(ch_1) \overrightarrow{}_{\text{snc}} \textbf{out}(ch, 0)_1 \parallel_{\hat{M}} \\ a_r = s \wedge \neg \textbf{wr}(ch_1) \wedge (b_1 \vee a_l = s) \overrightarrow{}_{\text{snc}} (a_r \mathrel{:=}_{\text{snc}} s_2)_1 \parallel_{\hat{M}} \\ a_r = s \wedge \neg \textbf{wr}(ch_1) \wedge \neg (b_1 \vee a_l = s) \overrightarrow{}_{\text{snc}} (a_r \mathrel{:=}_{\text{snc}} s)_1 \parallel_{\hat{M}} \\ a_r = s_1 \overrightarrow{}_{\text{snc}} (a_r \mathrel{:=}_{\text{snc}} f)_1 \parallel_{\hat{M}} a_r = s_2 \overrightarrow{}_{\text{snc}} (a_r \mathrel{:=}_{\text{snc}} f)_1 \end{array}\right), b_2]$$

Applying Lemma 6.2.5 and making

$$v = (\texttt{SELECT}\,(x, 0 \lhd (b_1 \wedge \neg \textbf{wr}(ch_1)) \rhd x, 0 \lhd (a_r = s \wedge \neg \textbf{wr}(ch_1)) \rhd x))$$

we obtain:

$$a_l, a_r : [b_1, \left(\begin{array}{l} b_1 \overrightarrow{}_{\text{snc}} (a_l, x \mathrel{:=}_{\text{snc}} (f, v) \lhd b_3 \rhd (s, x))_1 \parallel_{\hat{M}} \\ b_1 \wedge \mathbf{wr}(ch_1) \overrightarrow{}_{\text{snc}} (x, a_r \mathrel{:=}_{\text{snc}} \mathbf{in}(ch_1), s_1)_1 \parallel_{\hat{M}} \\ b_1 \wedge \neg\mathbf{wr}(ch_1) \wedge (b_1 \vee a_l = s) \overrightarrow{}_{\text{snc}} (a_r \mathrel{:=}_{\text{snc}} s_2)_1 \parallel_{\hat{M}} \\ b_1 \wedge \neg\mathbf{wr}(ch_1) \wedge \neg(b_1 \vee a_l = s) \overrightarrow{}_{\text{snc}} (a_r \mathrel{:=}_{\text{snc}} s)_1 \parallel_{\hat{M}} \\ a_r = s \overrightarrow{}_{\text{snc}} (a_l, x \mathrel{:=}_{\text{snc}} (f, v) \lhd b_3 \rhd (s, x))_1 \parallel_{\hat{M}} \\ a_r = s \wedge \mathbf{wr}(ch_1) \overrightarrow{}_{\text{snc}} (x, a_r \mathrel{:=}_{\text{snc}} \mathbf{in}(ch_1), s_1)_1 \parallel_{\hat{M}} \\ a_r = s \wedge \neg\mathbf{wr}(ch_1) \wedge (b_1 \vee a_l = s) \overrightarrow{}_{\text{snc}} (a_r \mathrel{:=}_{\text{snc}} s_2)_1 \parallel_{\hat{M}} \\ a_r = s \wedge \neg\mathbf{wr}(ch_1) \wedge \neg(b_1 \vee a_l = s) \overrightarrow{}_{\text{snc}} (a_r \mathrel{:=}_{\text{snc}} s)_1 \parallel_{\hat{M}} \\ a_r = s_1 \overrightarrow{}_{\text{snc}} (a_r \mathrel{:=}_{\text{snc}} f)_1 \parallel_{\hat{M}} a_r = s_2 \overrightarrow{}_{\text{snc}} (a_r \mathrel{:=}_{\text{snc}} f)_1 \end{array}\right), b_2]$$

Finally, applying Lemma 6.2.7 and predicate calculus we can update

$$v = (\texttt{SELECT}\,(\texttt{ARB}, 0 \lhd b_1 \rhd \texttt{ARB}, 0 \lhd a_r = s \rhd \texttt{ARB}))$$

also

$$b_3 = (b_1 \vee a_r = s)$$

and simplify the above formula to:

$$a_l, a_r : [b_1, \left(\begin{array}{l} b_1 \overrightarrow{}_{\text{snc}} (a_l, x \mathrel{:=}_{\text{snc}} (f, v) \lhd (b_1 \vee a_r = s) \rhd (s, x))_1 \parallel_{\hat{M}} \\ b_1 \wedge (b_1 \vee a_l = s) \overrightarrow{}_{\text{snc}} (a_r \mathrel{:=}_{\text{snc}} s_2)_1 \parallel_{\hat{M}} \\ b_1 \wedge \neg(b_1 \vee a_l = s) \overrightarrow{}_{\text{snc}} (a_r \mathrel{:=}_{\text{snc}} s)_1 \parallel_{\hat{M}} \\ a_r = s \overrightarrow{}_{\text{snc}} (a_l, x \mathrel{:=}_{\text{snc}} (f, v) \lhd (b_1 \vee a_r = s) \rhd (s, x))_1 \parallel_{\hat{M}} \\ a_r = s \wedge (b_1 \vee a_l = s) \overrightarrow{}_{\text{snc}} (a_r \mathrel{:=}_{\text{snc}} s_2)_1 \parallel_{\hat{M}} \\ a_r = s \wedge \neg(b_1 \vee a_l = s) \overrightarrow{}_{\text{snc}} (a_r \mathrel{:=}_{\text{snc}} s)_1 \parallel_{\hat{M}} \\ a_r = s_1 \overrightarrow{}_{\text{snc}} (a_r \mathrel{:=}_{\text{snc}} f)_1 \parallel_{\hat{M}} a_r = s_2 \overrightarrow{}_{\text{snc}} (a_r \mathrel{:=}_{\text{snc}} f)_1 \end{array}\right), b_2]$$

The condition in the third step is unfeasible (i.e., the whole step can be eliminated by means of Definition 3.3.149, Laws 3.3.25 and 3.3.145). It is also possible to simplify the condition in the second step and the actions in the first and fourth steps can be resolved using the information in the guard:

$$a_l, a_r : [b_1, \left(\begin{array}{l} b_1 \overrightarrow{\ }_{\text{snc}}(a_l, x \mathrel{:=}_{\text{snc}} f, 0)_1 \parallel_{\hat{M}} \\ b_1 \overrightarrow{\ }_{\text{snc}}(a_r \mathrel{:=}_{\text{snc}} s_2)_1 \parallel_{\hat{M}} \\ a_r = s \overrightarrow{\ }_{\text{snc}}(a_l, x \mathrel{:=}_{\text{snc}} f, 0)_1 \parallel_{\hat{M}} \\ a_r = s \wedge (b_1 \vee a_l = s) \overrightarrow{\ }_{\text{snc}}(a_r \mathrel{:=}_{\text{snc}} s_2)_1 \parallel_{\hat{M}} \\ a_r = s \wedge \neg(b_1 \vee a_l = s) \overrightarrow{\ }_{\text{snc}}(a_r \mathrel{:=}_{\text{snc}} s)_1 \parallel_{\hat{M}} \\ a_r = s_1 \overrightarrow{\ }_{\text{snc}}(a_r \mathrel{:=}_{\text{snc}} f)_1 \parallel_{\hat{M}} a_r = s_2 \overrightarrow{\ }_{\text{snc}}(a_r \mathrel{:=}_{\text{snc}} f)_1 \end{array}\right), b_2]$$

Finally, the first and second step can be merged into a single step (as they have the same guard), and steps three to five are unreachable (they can be eliminated by means of Laws 5.1.14, 5.1.13 and 3.3.99) leading us to the final normal form:

$$a_l, a_r : [b_1, \left(\begin{array}{l} b_1 \overrightarrow{\ }_{\text{snc}}(a_l, a_r, x \mathrel{:=}_{\text{snc}} f, f, 0)_1 \\ \parallel_{\hat{M}} a_r = s_2 \overrightarrow{\ }_{\text{snc}}(a_r \mathrel{:=}_{\text{snc}} f)_1 \end{array}\right), b_2]$$

Notice that:

1. Consistently with the semantics, the normal form performs the transmission over ch and the update of x with the value 0 during the first clock cycle. During the second clock cycle, the normal form just delays and terminates its execution.

2. The impossible communication over channel ch_1 did not lead to an infinite loop. In this case, the fact that the communication over ch_1 was inside a **priAlt** with other alternatives just leads to the elimination of the branches attempting to use ch_1.

3. The fact that all the information necessary to resolve the control path of the program was available in the source code allowed the simplification of the normal form and the elimination of unnecessary alternatives.

6.4 Chapter summary

In this chapter we have covered the following topics:

- **Algebraic reduction of the communication constructs to first normal form**. The reduction of input and output to first normal form is achieved by means of the basic communication primitives introduced in Chapter 4. The main advantage of this approach is the fact that it is fully algebraic and it allows reasoning at the right level of abstraction (i.e., still referring to channels rather than their implementation as wires and data buses).

- **Compilation of the priAlt construct to first normal form**. The reduction to first normal form is achieved by means of the basic communication primitives, the case expression introduced in Chapter 4 and their equivalence with the different forms of the **priAlt** construct.
- **Compilation of input,output and priAlt from first to second normal form**. Our approach for the transformation from first to second normal form in the context of communication and prioritised choice is to re-use the reduction strategy used for the other programming constructs. To make this possible, all primitive communication constructs and case statements are reduced to conditional assignments (the input to the reduction strategy to second normal form presented in Chapter 5).

Chapter 7

Conclusions

"*Begin thus from the first act, and proceed;*
and, in conclusion, at the ill which thou hast done,
be troubled, and rejoice for the good"

— Pythagoras

In this work we develop a verified compilation framework for the compilation of Handel-C [Celoxica Ltd. 2002*a*] into hardware using an algebraic approach. The compilation process is formalised within a new UTP [Hoare and He 1998] semantic framework: that of a synchronous, shared-variables language with capabilities for parallel behaviour and communications primitives. Within this framework, a formal model of the Handel-C is gradually transformed – by means of semantic-preserving reduction laws – into a formal abstraction of the target hardware components.

The process of reducing Handel-C into gate-level descriptions of hardware is characterised by a succession of transformations between different normal forms. As our target architecture is FPGA devices, there is no default structure or behaviour our normal forms are to mimic. Instead, we choose to design our normal forms to represent descriptions of state machines at different levels of abstraction. In this way, our final normal form is a low-level net-list description that can be implemented directly on an FPGA.

By formalising the compilation task as the successive reduction to different normal forms we have been able to capture the process in an incremental way using the normal forms as intermediate representations of the source code and identifying each of them with clearly defined compilation phases: the reduction of source programs into state machine and the simplification of parallel behaviour. One crucial aspect of our work is the fact that the first normal form allows the encoding of parallel behaviour in an a simple and efficient way. The efficiency in the encoding directly translates in a better applicability of our approach to industrial-scale projects, where the sheer size of the source code does not allow the limited resources of the FPGA to be wasted due to the compiler generating unnecessary hardware.

The simplicity of our first normal form, on the other hand, is the key feature that allows the simplification of parallelism performed in the second normal form. In particular, the fact that every computational step in our normal takes exactly one clock cycle and the regularity within a step

(i.e., a step is nothing else but a guarded assignment) is what allows otherwise infeasible reduction rules to hold in our context (e.g., the so called 'abides' property between $\|_{\hat{M}}$ and iteration as described in law 5.1.16). Encoding the program constructs in this way, however, forces the normal form to advance the clock cycle counter every time a change in control flow of the program is required. This restriction is the main reason why our compiler cannot handle iteration (remember our compilation technique only addresses a restricted form of iteration we call *iterated selection*).

Even though we have chosen Handel-C as the source language for our hardware compiler, we believe our source language comprises many features that are common to programming languages intended to be compiled into hardware. In fact, the kind of shared-variable parallelism used in Handel-C is much less restrictive, hence more complex to compile, than its counterparts in SystemC [Grotker 2002] or Verilog [Thomas and Moorby 1998]. From this observation, most of our compilation techniques can be directly re-used in the compilation of these languages. Furthermore, the uniform treatment of time we follow in this work makes our technique applicable to compile parallel, high level programming languages (e.g., Occam) directly into hardware.

The usual approach to algebraic compilation [Sampaio 1997] involves having a reasoning language as an algebraic structure whose axioms and reduction laws characterise the semantics of the language. We follow this approach as it has several advantages regarding modularity, simplicity and clarity. There is, however, the latent risk of postulating an incorrect axiom and allowing the possibility of proving the correctness of a faulty compiler. To avoid this problem, we have created a UTP [Hoare and He 1998] semantic model for languages with the programming features present in Handel-C (see Chapter 3). Our semantic model is not only the model and semantic foundation for our theory, but it is also the link allowing us to embed Handel-C within our reasoning language by providing a denotational semantics for Handel-C in terms of the reasoning language's constructs.

On top of the role in the context of this work, our theory of synchronous designs addresses one of the fields that, to the best of our knowledge, have not been covered with a dedicated UTP theory. Most efforts to address hardware [Butterfield 2007] or, in more general terms, synchronous systems [Pu et al. 2005; Butterfield et al. 2007] in the Unifying Theories of Programming have been based on the theory of reactive processes [Hoare and He 1998, Chapter 9]. In all of these works, synchronisation is controlled by means of processes entering a *waiting* state that is inherited from un-timed, non-determinism capable, process algebras like CSP, ACP or CCS. Even though this is a valid semantic framework for reasoning about synchronous systems, we believe that in fully deterministic contexts where the notion of time is more precise (e.g., as in the case of hardware-oriented programming languages), the semantic framework could benefit from this information in order to obtain a theory where it is simpler to reason and prove algebraic laws. Our theory of synchronous designs takes full advantage of this observation, allowing us to prove a comprehensive set of laws and theorems about synchronous systems.

The fact that our theory of synchronous designs is based on conjunctive healthiness conditions would allow to seamlessly extend it in order to accommodate additional programming features that already have their semantics formulated in terms of conjunctive healthiness conditions (e.g., the theory of pointers [Harwood et al. 2008; Smith and Gibbons 2008] or object oriented programming

[Smith and Gibbons 2007]).

Regardless of its comparison with other UTP theories, the theory of synchronous designs provides a very efficient way of describing (i.e., providing semantics) programming languages with a synchronous underlying model. In this regards, the high level of abstraction of the synchronous theory – achieved mainly through our extended parallel operator for processes with shared variables – allows the elimination of operational details, producing semantics that are better suited for reasoning and proving algebraic properties of the language.

The main weakness of the UTP and, consequently, of our theory, is its lack of support tools. As a consequence of this, all developments and proofs in this thesis have been carried out "by hand". On the other hand, the UTP itself together with all its algebraic reasoning laws makes this limitation a possibility for gaining further insight both in the theory being used and also in the model being constructed. Furthermore, the comprehensive support for programming constructs, together with the possibility of integrating additional programming features (by means of extending an existing theory or, as in our case, creating a completely new one) makes the approach more convenient in this context than other formalism with highly developed proving engines like HOL [Gordon and Melham 1993]. Finally, work has been carried out in order to provide mechanical support for the UTP by means of an embedding in ProofPower [Artan 2000] by Oliveira et al. [2006]; Zeyda and Cavalcanti [2009].

Regarding the design and construction of compilers using the algebraic approach, we believe the approach is not sufficiently mature yet to be a commercially viable solution. Although the mathematics involved in the task are not very deep, the amount of reasoning required to assert and show the correctness of a compiler might scare practitioners with a more practical background. A possible way forward is by having separate teams facing the task: one of computer scientists to continue the search for general theorems to support and enable the compilation of the missing features/programming languages; and another one of practitioners actually implementing this ideas in a suitable programming language.

Finally, the mapping from the actual source program into its model in the reasoning language (and, similarly, from the final normal form into the target architecture) cannot be formalised and, consequently, cannot be verified. This is a problem of provably correct systems in general, as there will always be a gap between any mathematical model and its implementation. Even if all the stages in the compilation process are proved correct, it is actually a mathematical model of them what is, in the end, verified. The purpose of pursuing formal verification is to gain a better insight into the problem and reduce the occurrence of errors, their total absence can never be proved.

This rest of this chapter summarises the main contributions of this thesis, its limitations and closely related research addressing how our work compares to other algebraic compilers for Hardware Description Languages. The chapter closes with a discussion of possibilities for future work and extensions.

7.1 Summary of contributions

We outline a summary of the contributions of this work below.

A theory of synchronous, parallel programs with shared variables. Our theory imposes additional restrictions (i.e., healthiness conditions) to the UTP theory of designs [Hoare and He 1998, Chapter 3] in order to make it suitable for reasoning about synchronous parallel programs with shared variables. The new theory is defined following the *conjunctive healthiness conditions* approach introduced by Cavalcanti and Woodcock [2006]. We have also added additional observational variables to record the progression of time and history of variables that enable the theory to handle synchronous behaviour in conjunction with shared variables. The most relevant features and contributions of the new theory are presented below.

- *Assertional reasoning over clock cycle boundaries*. The impossibility of asserting the value of variables after synchronisation points is one of the main limiting factors that pre-empts the usage of the simpler theory of designs as the semantic framework for our work. Our additional healthiness conditions and treatment of synchronicity allow us to prove that

  $$(x \overset{:=}{_{\mathsf{snc}}} e; \mathsf{sync}) \quad = \quad x \overset{:=}{_{\mathsf{snc}}} e; \mathsf{sync}; (x = e)^{\perp}_{\mathsf{S}}$$

  when there is no other process modifying x during that clock cycle[1]. This is one of the key results in order to be able to show the correctness of the compilation theorems.

- *A different-length, parallel-by-merge operator*. A common feature of the parallel-by-merge operators in the UTP theories is that they can only handle parallel processes that take the same amount of clock-cycles to terminate. This is an unreasonable restriction in the context of Hardware Description Languages, where any pair of arbitrary programs (even non-terminating ones) can be combined in parallel. Our theory solves this problem by introducing a new merge predicate that accounts for the difference in duration of the parallel processes while satisfying the same set of properties of the original UTP merge operator.

- *Dynamic scope and alphabet extension in a context of variables with history*. The inclusion of a mechanism to record the variable's history hinders the applicability of the UTP operators used to control the scope of variables. We include new operators for dynamic scope and alphabet extension that account not only for the history of variables but also for the additional healthiness conditions in our theory. From the reasoning perspective, the need for these new definitions makes the extension of a variable's scope lead to refinement when it is done over clock-cycle boundaries.

- *Algebraic characterisation of programming constructs and operators*. As shown in Chapters 5 and 6, the set of algebraic laws presented in our theory is sufficient to reason about syn-

[1] Note that this law does not hold when multiple processes are trying to modify x simultaneously. For an example, consider the case of $(x \overset{:=}{_{\mathsf{snc}}} e_1; \mathsf{sync}) \parallel_{\hat{M}} (x \overset{:=}{_{\mathsf{snc}}} e_2; \mathsf{sync})$ where it is not clear what the value of x should be after the parallel assignment.

chronous programs with shared variables. Furthermore, it is possible to use the synchronous theory as the underlying formalism in which to give semantics to Handel-C programs.

In comparison with the UTP theory of designs, most properties about the programming operators hold in the same form or as refinements in our new theory. On the other hand, the fact that our theory keeps track of the history of variables automatically invalidates the property that

$$P; \bot = \bot$$

(abort is a right zero for sequential composition). In consequence, all properties about designs that depended on this condition (e.g., disjoint-alphabet parallelism can be expressed as sequential composition) do not hold in our theory.

For our algebraic Handel-C compiler, the theory of synchronous designs acts as the semantic foundation for our reasoning language. In this way, we are not only able to show the validity of the basic axioms in the reasoning language, but we can also ensure that there is at least one model that can satisfy them.

UTP semantics for Handel-C. Our semantics is based on the theory of synchronous designs described in Chapter 3. Our Handel-C semantics is an improvement on other denotational semantics of the language [Butterfield 2007; Butterfield et al. 2007; Butterfield and Woodcock 2006; 2005*a*] regarding their simplicity towards the discovery/proof of algebraic laws. Other semantics are either based on ad-hoc formalisms not suitable for proving algebraic properties or are based on a deep embedding of state machines in the UTP reactive theory, which introduces nondeterministic behaviour and refusals that are not necessary in the context of Handel-C[2]. The main contribution of our UTP semantics for Handel-C are as follows:

- *Algebraic laws and equivalence relationships proved about Handel-C programs*. The set of algebraic laws about Handel-C serves this work in two ways: (i) it allows to prove equivalences between different constructs and operators that simplify the compilation into hardware, as described in Chapter 6; and (ii) it provides reassurance on the correctness of the semantics, as the results captured by the laws are consistent with the expected behaviour of Handel-C.

- *A formal link between Handel-C and our reasoning language*. Since Handel-C and the reasoning language share the same semantic domain (i.e., the theory of synchronous designs) it is straightforward to show that our source language is embedded within our reasoning language. There is evidence in the literature of this kind of link between denotational models and refinement frameworks for the same language [Cavalcanti and Naumann 2000], yet this is the first time the UTP has been used as the semantic foundation for an algebraic compiler.

[2] As Handel-C is a programming language with no constructs for program specification, it does not contain the nondeterministic choice operator; thus there is no need to account for this construct and the mathematical elements involved in its semantics (i.e., refusals).

- *Characterisation of the communication primitives and **priAlt** in terms of more basic operators*. The main advantage of our decomposition of these constructs is that it allows their treatment via algebraic laws that still operate over channels without the need to resort to lower-level implementation details.

Verified hardware synthesis for Handel-C. We show how the algebraic approach to compilation is used to produce a correct-by-construction compiler for Handel-C into hardware. The main contributions of this part of the thesis are as follows:

- *Highly parallel normal form*. Our normal form is based on the idea of a machine executing one-clock-cycle steps. The way in which steps are combined allows more than one of them to be activated at any given clock cycle. Parallelism is captured directly at the control-flow level of our normal form; hence minimising the hardware required to implement parallel behaviour. Other hardware compilation approaches are based on normal forms that address parallel behaviour by means of generating particular forms of the cross product among the machines that are composed in parallel. This is highly inefficient as it produces hardware for all possible interactions between the two machines, even those combinations that are not possible. Our approach is more suitable than these cross-product-based techniques, if we consider that the target of our compiler is FPGA devices, where the size of the generated hardware is crucial.

- *Verified synthesis for parallel behaviour with shared variables*. To the best of our knowledge, all existing works in the literature either do not address this problem at all, or take place in contexts where the alphabet is partitioned among parallel processes (e.g., Occam).

- *Algebraic compilation of **priAlt** and communications in a shared-variables context*. We provide a fully compositional treatment of these constructs and ways of simplifying the communications patterns in certain cases where the program contains enough information to do so. A distinguishing feature of our work is the fact that all communication primitives are transformed into conditional assignments when the program reaches its final normal form; hence simplifying and minimising the hardware necessary to implement these constructs.

- *The reduction of Handel-C programs to a single assignment*. If we consider the constructs in Handel-C it is easy to see that all programs that can be generated are, in essence, sequences of assignments. In this context, the control-flow of a Handel-C program can be seen as a selection mechanism deciding which assignment is performed at which clock-cycle. It is a reassuring result that our compiler can derive this underlying essence of Handel-C programs whilst compiling them into hardware.

- *Simpler and more efficient (in terms of resource utilisation) generated code before optimisations when compared to commercial alternatives*. If we compare the before-optimisations results of our compiler with the output of the commercial compiler for Handel-C [Celoxica Ltd. 2002*b*] it is clear that our code: (a) has a much simpler structure, as the generated hardware by commercial tools follows the imperative-structure of the original Handel-C

program; and (b) is more compact, as the timing in our synthesised hardware is controlled implicitly by the updates to registers performed in our single-assignment normal form. Regarding (b), the code generated by commercial tools requires the introduction of additional flip-flops in order to maintain the timing and synchronicity of the generated code. In the hardware generated with our approach, time and synchronicity is controlled implicitly by the fact that control variables are stored in flip-flops and the behaviour of the whole program is captured as the iteration of a conditional one-clock-cycle assignment.

7.2 Limitations

The compilation of synchronous, parallel programs with shared-variables is a complex task, especially when it is performed with the additional goal of a formal proof of correctness in mind. Our approach goes a long way towards overcoming many of those difficulties, yet there are limitations to the results of this thesis.

Multi-clock-cycle expression evaluation. One of the assumptions of our work is that all expressions can be evaluated by means of combinatorial logic within a single clock-cycle. This is the case for all expressions involving addition, subtraction and logical operations. Other more complex operations, like multiplication and division, can still be performed within a single clock-cycle but they require the frequency of the clock to be significantly reduced. In commercial applications, expressions involving these time-consuming operations are split across several clock-cycles by means of pipelined implementations. Unfortunately, complex data-flow analysis is required in order to be able to correctly switch to this more efficient implementation and our compilation approach is currently not capable of performing them. In case performance is a crucial factor, a Handel-C description of a pipelined implementation of these operations may be a suitable alternative, at the expense of the user performing all the data dependency analysis at the source-code level.

Suboptimal implementation of the SELECT function that selects the values to update the program/control variables. This is not a limitation arising from our mapping of the normal form into hardware, but a consequence of the definition of the merge predicate in the semantics of our reasoning language. If one analyses our definition of the SELECT function (see Section 2.3.8.1), it is possible to find it requires the comparison of the values held in a variable with the possible values to update it. This is the mechanism we use in the semantics to detect whether any of the processes writing to a variable x had modified it during the current clock-cycle. On the other hand, in the second normal form we have expressions of the form

$$((w_1 \overset{:=}{_{\text{snc}}} e_1 \lhd c_1 \rhd x) \parallel (w_2 \overset{:=}{_{\text{snc}}} e_2 \lhd c_2 \rhd x) \dots) ; (x \overset{:=}{_{\text{snc}}} \texttt{SELECT}\,(x, w_1, w_2))_1$$

where w_1 will carry an update value for x only when condition c_1 holds. In this context, there is no need for the comparison between w_1 and x, and we could simply implement the expression above

as

$$((w_1 \underset{\text{snc}}{:=} e_1) \parallel (w_2 \underset{\text{snc}}{:=} e_2) \dots) ; (x \underset{\text{snc}}{:=} (e_1 \lhd c_1 \rhd (e_2 \lhd c_2 \rhd x)))_1$$

The second expression is a more efficient way of achieving the same result, as the selection of values to update x is not based on comparing the value of x with w_1 and w_2, but on simply evaluating a condition. Unfortunately, due to the way assignment is defined in our theory, it is not possible to derive the above simplification in the current state of the semantics for our reasoning language. In order to improve this aspect of the thesis, our semantics would need to be modified in order to have a more concrete way of determining whether a process has changed the value of a variable or not. Due to time restrictions and the limited impact this modification will have towards the main objective of this thesis (i.e., correctness of the compilation), we regard its solution as a future extension of our work.

Treatment of the while construct. In order to be able to compile programs into normal form we impose that all **while** constructs must be followed by at least one clock-cycle consuming statement. As mentioned in Section 5.1.4.5, this is a restriction arising from the way we encode combinatorial changes in the control-flow within our first normal form. The user is required to find a suitable part of the program (or add a **delay** construct) that can be executed after each **while** construct that does not satisfy the above criteria. This change potentially leads to decreased efficiency in the whole program. In this direction: (a) our pilot study suggests that these changes are rarely necessary in most programs (see Section 5.1.4.4 for further details); and (b) we expect our compiler to be used in contexts where the emphasis is on program correctness and a minor loss in performance can be tolerated in exchange for a correct outcome from the compiler.

7.3 Closely related work

We have mentioned and briefly described related work throughout the different chapters of this thesis. In this section we focus on closely related work and consider their differences with our approach. There have been two major research projects on applying the algebraic technique to compile both Occam and Verilog into hardware.

7.3.1 Occam

In [He et al. 1992; 1993; Bowen et al. 1994], He and others propose algebraic transformation rules to transform a subset of Occam into a normal form suitable for an implementation in hardware. This seminal work addresses a subset of Occam including communications, parallelism and alternation. The compilation down to hardware resembles our approach, as their normal form is based on control and state variables. Two mappings are stored within the normal form: one determining the next state based on the current one; and another one associating the current control state to the value for the program and control variables. Their hardware-generation phase follows a similar approach to ours as well. Latches (i.e., memory) are allocated to store the control and state

variables, while the mappings are implemented by combinatorial means. In comparison with our work, the main differences lay in the following facts:

The lack of a time model for Occam. This allowed He and Paige to define a time model where conditions take a whole clock-cycle to be computed. In this context, their state-based normal form encoding becomes much simpler, as they do not need to deal with conditionals being resolved in combinatorial time. This approach is not an option in our context as the timing of the constructs is defined by Handel-C semantics, which is different from Occam's, and cannot be altered without leading to incorrect behaviour in the generated hardware.

On the other hand, He, Bowen and Paige also include additional laws to address the control of time and to introduce assumptions and assertions into the language. In line with the concepts in our work, the clock-cycle counter is controlled by means of the action $I\!I_1$ (the action of taking one clock cycle and keeping all variables constant). Nevertheless, they also postulated

$$I\!I \sqsubseteq I\!I_1$$

as one of their basic laws. In a context of shared variables, this law could lead to inconsistent behaviour due to the possible change in the timing of the program. For example, consider the fragment

$$ch?x \parallel ch!e$$

which is equivalent to the assignment $x := e$. However, we can apply the refinement law above together with the fact that $\boldsymbol{I}$ is the left unit for sequential composition in Occam to refine the original program into the equivalent form

$$ch?x \parallel (I\!I_1 ; ch!e)$$

In Occam, this transformation is semantics-preserving because there are no shared variables and; thus, the effects of delaying the execution of one of the parallel branches has no effects regarding its environment. On the other hand, this way of manipulating the time structure of the program is not possible in the context of Handel-C programs, as a delayed action in one of the parallel branches may lead to a change in the control flow in the others.

The lack of shared-variables in Occam. The main consequence of this feature is the additional freedom regarding the timing of the program mentioned above. On the other hand, this feature of Occam also allows a much simpler algebraic treatment of parallel programs due to the more comprehensive set of laws made available to the compilation process. The fact that variables are not shared among parallel processes makes it possible to:

- *Eliminate the need to keep track of the history of variables*. Avoiding this additional complication (necessary in our context) increases the number of properties of the whole set of constructs in the reasoning language. For example, the law stating that $\perp$ is the right zero

for sequential composition (i.e., $(P; \bot) = \bot$) can only be proved in a context where history is not recorded.

- *Have a simpler parallel operator.* In algebraic terms, this means the possibility of executing parallel processes concurrently or by any possible interleaving of their actions. Furthermore, as there is no separation of variables or merging at the end of the parallel operator, $I\!I$ is the unit for parallel composition (i.e., $(I\!I \parallel P) = P$); hence allowing a simple treatment of parallel behaviour when the different branches take different amounts of clock-cycles to terminate.

Encoding of parallelism in the normal form. As mentioned in Section 7.1, their approach to compiling parallel composition is to construct the "product" machine of the two normal forms (i.e., to calculate all possible control combinations when the two normal forms execute concurrently). Even though this is an elegant mathematical solution to the problem, it is inefficient in terms of resource utilisation as the approach must cover all possible combinations of control states, even if they will never be executed. This is not a feasible solution in the context of our work, where the target are highly-restricted devices in terms of available hardware resources.

Reasoning language. Their reasoning language is a superset of the programming laws for Occam proposed by Roscoe and Hoare [1988] and differs from ours in a number of ways, mainly regarding the algebraic treatment of miracle within parallel contexts. When analysing these differences we found a few conflicting laws that may lead to problems when using them as the reasoning language for the compiler. For instance, the law

$$I\!I \parallel (ch?x; P) = \textbf{stop}$$

suggests a treatment of parallelism similar to [Hoare 1983] where parallel processes are forced to have the same alphabets and to synchronise on all of their variables. On the other hand, Occam does not have a notion of alphabets associated to processes and, moreover, it only allows a single pair of input/output commands in the whole program. In the light of this observation, let us consider $R = (ch!v; S)$ then

$$(I\!I \parallel (ch?x; P)) \parallel R$$

is not equivalent to

$$I\!I \parallel ((ch?x; P) \parallel R)$$

contradicting the associative property of parallel composition in Occam [Roscoe and Hoare 1988]. It is also not clear how the authors guarantee that their parallel operator allows processes of different length when they provide a spreadsheet principle similar to the one presented in our work yet their parallel operator does not have $I\!I$ as a unit.

In summary, there are major differences between Occam and Handel-C (mainly due to an unclear time model and the lack of shared variables) that would make the techniques developed by He and others [He et al. 1992; 1993; Bowen et al. 1994] unsuitable in our context. Furthermore, some of the basic compilation laws for Occam do not hold in our context of synchronous, parallel programs with shared-variables (for example, abort being the right zero for sequential composition). Finally, our work provides an additional level of confidence regarding its correctness as we have created a denotational model (see Chapter 3) from where we have proved all of our reasoning laws.

7.3.2 Verilog

Another approach towards hardware synthesis based on the algebraic approach was proposed by Iyoda and He [Iyoda and He 2001*b*; He 2002]. In this work, the authors describe an axiomatic approach towards the compilation of Verilog programs into hardware descriptions.

The first step, reduction to assignment normal form, consists of the normalisation of assignment by transforming each individual variable assignment into a total assignment (i.e., an assignment that updates the value of the variable being modified and keeps constant the rest of identifiers in the program). Rules are also provided in order to simplify assignments when combined by sequential composition or conditionals.

The second step, reduction to conditional normal form, provides rules in order to deal with variable's values being preserved over clock-cycle edges (i.e., variables that cannot be synthesized into combinatorial logic but into memory-based components). These rules capture the notion of execution paths occurring during the program execution by using conditional constructs, highlighting the variables that are not modified along them.

Finally, rewriting rules are applied after these two stages in order to perform the transformation into Xilinx's net-list format (XNF). This step consists, mainly, in mapping all expressions into net-list form by defining their operators as symbols and assigning the involved names to wires in the generated hardware. The approach has also been mechanized in Prolog [Iyoda and He 2001*a*] and also extended to cope with hardware/software partitioning [Qin et al. 2002].

In comparison with our work, Iyoda and He address a small subset of Verilog that does not include iteration. On top of this, Verilog does not allow shared variables, leading to the same kind of simplifications applied when compiling Occam into Hardware. Finally, the wires and connection schema among different hardware components are not derived algebraically but produced by the above mentioned production rules, making the verification of this aspect of their work less formal.

7.4 Future work

The research presented in this thesis can be developed and extended along the following lines:

Extending the translatable subset of Handel-C. There are two major kinds of constructs in Handel-C that have been left out of this work: pointers and hardware optimisation keywords. Harwood and others [Harwood et al. 2008] have proposed a UTP theory of pointers based on

conjunctive healthiness conditions. This could be taken as the starting point to extend our semantic framework and the reasoning language to incorporate pointers and pointer arithmetic to our input language.

Regarding hardware optimisation keywords, these are usually carried along with the code and passed as additional constraints to the optimisation/place & route stages. In this context, the reasoning language will have to be extended with additional constructs in order to incorporate this kind of information for later stages to take advantage of it.

Algebraic treatment of optimisations and place & routing. A natural extension of our work towards a complete (and industrially attractive) tool chain from Handel-C down to FPGAs is to add correct implementations of the optimisation and place & route routines. From the point of view of the optimiser, a whole set of rules should be added to allow the simplification of our final normal form to eliminate: (a) redundant hardware; (b) allocation of RAM when possible (the above mentioned extension of the reasoning language to carry this information along the compilation process will be necessary in this context); (c) splitting of the calculation of multi-cycle expressions (e.g., those involving multiplication or division) into a pipelined implementation.

In turn, the implementation of place & route algorithms is highly dependant of the specific FPGA that will be programmed with the output of our compiler. This means a verified 'back-end' for our compiler will have to be implemented for each specific architecture that needs to be addressed as a target. The usage of heuristics is very common at this stage, as some of the routing problems among different components can be NP-complete; hence it cannot be addressed with standard techniques. In this sense, the hardware/software co-design described by Silva et al. [1997*a*; 2004] is an interesting initial point to address these aspects within the algebraic framework.

Appendices

Appendix A

Proofs from Chapter 2

This appendix provides the details about the proofs of the laws and theorems stated in chapter 2. Note that as chapter 2 intends to provide an overview of the UTP, some of the results presented there are taken directly from the work by Hoare and He [1998]. Hence, the proofs we provide here are of those laws and theorems that have been added to the original UTP theory for this thesis. Results that have been taken directly from the UTP are not proved here, as the proofs are available in [Hoare and He 1998; Woodcock and Cavalcanti 2004].

Law 2.2.11 $P \sqcap Q \sqsubseteq P$

Proof.

$$
\begin{aligned}
& [P \Rightarrow P] \\
\Rightarrow & \{\text{Propositional calculus}\} \\
& [P \Rightarrow (P \vee Q)] \\
\equiv & \{\text{Definition 2.2.9}\} \\
& [P \Rightarrow P \sqcap Q] \\
= & \{\text{Definition 2.2.10}\} \\
& P \sqsupseteq P \sqcap Q
\end{aligned}
$$

□

Law 2.3.13 $x := e; \top_D = \top_D$

Proof.

$$
\begin{aligned}
& x := e; \top_D \\
= & \{\text{Theorem 2.3.5, definition 2.3.9, theorem 2.3.19}\} \\
& (\mathbf{true} \wedge \neg(x := e; \neg\mathbf{true})) \vdash (x := e; \mathbf{false}) \\
= & \{\text{Definitions 2.2.6 and 2.2.3, propositional calculus}\} \\
& (\mathbf{true} \wedge \neg\mathbf{false} \vdash \mathbf{false}) \\
= & \{\text{Propositional calculus, theorem 2.3.5}\}
\end{aligned}
$$

$\top_D$ □

Law 2.3.14 $x := e; \mathit{II}_D = x := e$

Proof.

$x := e; \mathit{II}_D$
= {Definitions 2.3.9 and 2.3.8}
$(\mathbf{true} \vdash x := e); (\mathbf{true} \vdash \mathit{II})$
= {Theorem 2.3.19 and propositional calculus}
$(\mathbf{true} \vdash x := e; \mathit{II})$
= {Law 2.2.8, definition 2.3.9}
$x := e$ □

Law 2.3.30 $(P \lhd b \rhd Q) \lhd c \rhd R = (P \lhd c \rhd R) \lhd b \rhd (Q \lhd c \rhd R)$

Proof.

$(P \lhd b \rhd Q) \lhd c \rhd R$
= {Laws 2.3.23 and 2.3.29}
$(R \lhd \neg c \rhd P) \lhd b \rhd (R \lhd \neg c \rhd Q)$
= {Law 2.3.23}
$(P \lhd c \rhd R) \lhd b \rhd (Q \lhd c \rhd R)$ □

Law 2.3.32 $P \lhd (s \wedge b) \rhd (Q \lhd (s \wedge \neg b) \rhd R) = P \lhd (s \wedge b) \rhd (Q \lhd s \rhd R)$

Proof.

$P \lhd (s \wedge b) \rhd (Q \lhd (s \wedge \neg b) \rhd R)$
= {Laws 2.3.23 and 2.3.28}
$Q \lhd s \wedge \neg b \wedge \neg(s \wedge b) \rhd (R \lhd \neg(s \wedge b) \rhd P)$
= {Propositional calculus $((s \wedge \neg b \wedge \neg(s \wedge b)) = (s \wedge \neg b))$}
$Q \lhd (s \wedge \neg b) \rhd (R \lhd \neg(s \wedge b) \rhd P)$
= {Propositional calculus $((s \wedge \neg(s \wedge b)) = (s \wedge \neg b))$}
$Q \lhd s \wedge \neg(s \wedge b) \rhd (R \lhd \neg(s \wedge b) \rhd P)$
= {Law 2.3.28}
$(Q \lhd s \rhd R) \lhd \neg(s \wedge b) \rhd P$
= {Law 2.3.23}
$P \lhd (s \wedge b) \rhd (Q \lhd s \rhd R)$ □

Law 2.3.49 Provided that P is **H3** and e does not mention x we have that:

$$(P; x := e) = ((P; \textbf{end}\ x)_{+x}; x := e)$$

Proof.

$$\begin{array}{ll}
& P; x := e \\
= & \{\text{Theorem 2.3.19, } P \text{ is } \textbf{H3}\text{, propositional calculus}\} \\
& (P_1 \vdash P_2; x := e) \\
= & \{\text{Definition 2.2.3 and assumption: } x \text{ is not free in } e\} \\
& (P_1 \vdash \exists x_0, y_0 \bullet P_2[x_0, y_0/x', y']; x := e[y_0/y']) \\
= & \{x_0 \text{ is not free in } x := e\} \\
& (P_1 \vdash \exists y_0 \bullet (\exists x_0 \bullet P_2[x_0, y_0/x', y']); x := e[y_0/y']) \\
= & \{\text{Change bound variable, restrict scope}\} \\
& (P_1 \vdash \exists y_0 \bullet (\exists x' \bullet P_2)[y_0/y']; x := e[y_0/y']) \\
= & \{\text{Definition 2.2.19}\} \\
& (P_1 \vdash \exists y_0 \bullet (P_2; \textbf{end}\ x)[y_0/y']; x := e[y_0/y']) \\
= & \{\text{One-point rule}\} \\
& (P_1 \vdash \exists y_0, x_0 \bullet x_0 = x \wedge (P_2; \textbf{end}\ x)[y_0/y']; x := e[y_0/y']) \\
= & \{\text{Substitution}\} \\
& (P_1 \vdash \exists y_0, x_0 \bullet (x' = x \wedge (P_2; \textbf{end}\ x)[y_0/y'])[x_0/x']; x := e[y_0/y']) \\
= & \{\text{Substitution}\} \\
& (P_1 \vdash \exists y_0, x_0 \bullet (x' = x \wedge P_2; \textbf{end}\ x)[y_0, x_0/y', x']; x := e[y_0/y']) \\
= & \{\text{Definition 2.3.46}\} \\
& (P_1 \vdash \exists y_0, x_0 \bullet (P_2; \textbf{end}\ x)_{+x}[y_0, x_0/y', x']; x := e[y_0/y']) \\
= & \{\text{Theorem 2.3.19}\} \\
& (P; \textbf{end}\ x)_{+x}; x := e
\end{array}$$

□

Law 2.3.56 $b^\top; b^\top = b^\top$

Proof. Straightforward from law 2.3.60 and idempotence of conjunction. □

Law 2.3.57 $b_\bot; b_\bot = b_\bot$

Proof. Direct from law 2.3.61 and idempotence of conjunction. □

Law 2.3.58 $b_\bot = b_\bot; b^\top$

Proof.

$$\begin{array}{ll}
& b_\bot; b^\top
\end{array}$$

= {Theorem 2.3.52 and 2.3.53}

$(I\!I_D \lhd b \rhd \bot_D); (I\!I_D \lhd b \rhd \top_D)$

= {Laws 2.3.24, 2.3.10, 2.3.4}

$(I\!I_D \lhd b \rhd \top_D) \lhd b \rhd \bot_D$

= {Law 2.3.27 and theorem 2.3.53}

$b_\bot$ □

Law 2.3.59 $b^\top ; b_\bot = b^\top$

Proof.

Similar to the proof of law 2.3.58. □

Theorem 2.3.52 Design characterisation of assumption

$$b^\top = (\mathbf{true} \vdash I\!I \lhd b \rhd \mathbf{false})$$

Proof.

$b^\top$

= {Definitions 2.3.50, 2.3.8 and theorem 2.3.5}

$(\mathbf{true} \vdash I\!I) \lhd b \rhd (\mathbf{true} \vdash \mathbf{false})$

= {Theorem 2.3.18}

$(\mathbf{true} \lhd b \rhd \mathbf{true} \vdash I\!I \lhd b \rhd \mathbf{false})$

= {Law 2.3.22, definition 2.2.13}

$(\mathbf{true} \vdash I\!I \lhd b \rhd \mathbf{false})$ □

Theorem 2.3.53 Design characterisation of assertion

$$b_\bot = (b \vdash I\!I)$$

Proof.

$b_\bot$

= {Definitions 2.3.51, 2.3.8 and theorem 2.3.3}

$(\mathbf{true} \vdash I\!I) \lhd b \rhd (\mathbf{false} \vdash \mathbf{true})$

= {Theorem 2.3.18}

$(\mathbf{true} \lhd b \rhd \mathbf{false} \vdash I\!I \lhd b \rhd \mathbf{true})$

= {Definition 2.2.1, propositional calculus}

$((b \vee \mathbf{false}) \vdash b \wedge I\!I)$

= {Propositional calculus, definition 2.3.1}

$ok \wedge b \Rightarrow ok' \wedge b \wedge I\!I$

= {Propositional calculus, definition 2.3.1}

$(b \vdash I\!I)$ □

Law 2.3.60 $b^\top ; c^\top = (b \wedge c)^\top$

Proof.

$b^\top ; c^\top$

= {Theorem 2.3.52}

$(I\!I_D \lhd b \rhd \mathbf{false}) ; (I\!I_D \lhd c \rhd \mathbf{false})$

= {Laws 2.3.24,2.3.34,2.3.10}

$(I\!I_D \lhd c \rhd \mathbf{false}) \lhd b \rhd \mathbf{false}$

= {Law 2.3.28}

$I\!I_D \lhd c \wedge b \rhd (\mathbf{false} \lhd b \rhd \mathbf{false})$

= {Law 2.3.20 and theorem 2.3.50}

$(b \wedge c)^\top$ □

Law 2.3.61 $b_\perp ; c_\perp = (b \wedge c)_\perp$

Proof.

$b_\perp ; c_\perp$

= {Theorem 2.3.53}

$(b \vdash I\!I) ; (c \vdash I\!I)$

= {Theorem 2.3.19}

$(\neg(\neg b ; \mathbf{true}) \wedge \neg(I\!I ; \neg c) \vdash I\!I ; I\!I)$

= {Definition 2.2.3, propositional calculus and law 2.3.10}

$(b \wedge c \vdash I\!I)$

= {Theorem 2.3.53}

$(b \wedge c)_\perp$ □

Law 2.3.62 $(b \vee c)^\top ; b^\top = b^\top$

Proof. Trivial from law 2.3.60 and absorption of conjunction. □

Law 2.3.63 $b^\top ; c^\top = c^\top ; b^\top$

Proof. Trivial from property 2.3.60 and commutativity of conjunction. □

Law 2.3.64 $b^\top \sqsupseteq I\!I_D \sqsupseteq b_\perp$

Proof.

$$
\begin{array}{ll}
& b^\top \\
= & \{\text{Theorem 2.3.52}\} \\
& (\mathbf{true} \vdash I\!I \lhd b \rhd \mathbf{false}) \\
\sqsupseteq & \{\text{Law 2.2.15}\} \\
& (\mathbf{true} \vdash I\!I \lhd b \rhd I\!I) \\
= & \{\text{Law 2.3.22 then definition 2.3.8}\} \\
& I\!I_D \\
= & \{\text{Law 2.3.22}\} \\
& I\!I_D \lhd b \rhd I\!I_D \\
\sqsupseteq & \{\text{Law 2.2.14}\} \\
& I\!I_D \lhd b \rhd \mathbf{true} \\
= & \{\text{Definition 2.3.51}\} \\
& b_\perp
\end{array}
$$

□

Law 2.3.67 If b does not depend on x then $x := e; b^\top = b^\top; x := e$

Proof.

$$
\begin{array}{ll}
& x := e; b^\top \\
= & \{\text{Definition 2.3.50}\} \\
& x := e; (I\!I_D \lhd b \rhd \top_D) \\
= & \{\text{Laws 2.3.25, 2.3.14, 2.3.13}\} \\
& x := e \lhd b[e/x] \rhd \top_D \\
= & \{\text{Assumption } (b \text{ does not mention } x), \text{ laws 2.3.34, 2.3.10 and 2.3.24}\} \\
& (I\!I \lhd b \rhd \top_D); x := e \\
= & \{\text{Definition 2.3.50}\} \\
& b^\top; x := e
\end{array}
$$

□

Law 2.3.68 If b does not depend on x then $x := e; b_\perp = b_\perp; x := e$

Proof.

Similar to the proof of 2.3.67 but using law 2.3.12 instead of 2.3.13. □

Law 2.3.69 Provided that P is **H3** and **H4** we have that:

$$P_{+x}; (x = e)_\perp = (x = e)_\perp; P_{+x}$$

Proof.

$$
\begin{aligned}
& P_{+x}; (x = e)_\perp \\
= \; & \{P \text{ is a design, definition 2.3.46 and theorem 2.3.53}\} \\
& (P_1 \vdash P_2 \wedge x' = x); (x = e \vdash I\!I) \\
= \; & \{\text{Theorem 2.3.19 and assumption } (P \text{ is } \mathbf{H3})\} \\
& (P_1 \wedge \neg(P_2 \wedge x' = x; \neg(x = e)) \vdash P_2 \wedge x' = x; I\!I) \\
= \; & \{\text{Definition 2.2.3 and predicate calculus}\} \\
& (P_1 \wedge \neg(P_2 \wedge \neg(x = e); \mathbf{true}) \vdash P_2) \\
= \; & \{\neg(x = e) \text{ does not mention dashed variables, predicate calculus}\} \\
& (P_1 \wedge \neg(\neg(x = e) \wedge (P_2; \mathbf{true})) \vdash P_2 \wedge x' = x) \\
= \; & \{\text{Propositional calculus}\} \\
& ((P_1 \wedge x = e) \vee (P_1 \wedge \neg(P_2; \mathbf{true})) \vdash P_2 \wedge x' = x) \\
= \; & \{\text{Propositional calculus}\} \\
& ((P_1 \wedge x = e) \vee \neg(P_1 \Rightarrow (P_2; \mathbf{true})) \vdash P_2 \wedge x' = x) \\
= \; & \{\mathbf{H4}(P_1 \vdash P_2) \equiv [P_1 \vdash (P_2; \mathbf{true})], \text{propositional calculus}\} \\
& ((P_1 \wedge x = e) \vdash P_2 \wedge x' = x) \\
= \; & \{\text{Propositional calculus, law 2.3.10}\} \\
& ((x = e \wedge \neg(I\!I; \neg P_1) \vdash I\!I; (P_2 \wedge x' = x)) \\
= \; & \{\text{Theorem 2.3.19 and propositional calculus}\} \\
& (x = e \vdash I\!I); (P_1 \vdash P_2 \wedge x' = x) \\
= \; & \{P \text{ is a design, definition 2.3.46 and theorem 2.3.53}\} \\
& (x = e)_\perp; P_{+x}
\end{aligned}
$$

□

Law 2.3.70 $x := y; (y = e)_\perp = x := y; (x = e)_\perp$

Proof.

$$
\begin{aligned}
& x := y; (y = e)_\perp \\
= \; & \{\text{Theorem 2.3.53 and law 2.3.25}\} \\
& (x := y; I\!I_D) \lhd (y = e)[y/x] \rhd (x := y; \perp_D) \\
= \; & \{\text{Propositional calculus}\} \\
& (x := y; I\!I_D) \lhd (x = e)[y/x] \rhd (x := y; \perp_D) \\
= \; & \{\text{Inverse steps}\} \\
& x := y; (x = e)_\perp
\end{aligned}
$$

□

Law 2.3.71 $P \lhd b \rhd Q = (b^\top; P) \lhd b \rhd ((\neg b)^\top; Q)$

Proof.

$$
\begin{array}{ll}
 & (b^\top ; P) \lhd b \rhd ((\neg b)^\top ; Q) \\
= & \{\text{Definition 2.3.50, laws 2.3.24, 2.3.10 and 2.3.34}\} \\
 & (P \lhd b \rhd \top_D) \lhd b \rhd (Q \lhd \neg b \rhd \top_D) \\
= & \{\text{Law 2.3.27 (twice)}\} \\
 & P \lhd b \rhd Q
\end{array}
$$

□

Law 2.3.72 $b^\top ; (P \lhd b \rhd Q) = b^\top ; P$

Proof.

$$
\begin{array}{ll}
 & b^\top ; (P \lhd b \rhd Q) \\
= & \{\text{Definition 2.3.50, laws 2.3.24, 2.3.10 and 2.3.34}\} \\
 & (P \lhd b \rhd Q) \lhd b \rhd \top_D \\
= & \{\text{Laws 2.3.27, 2.3.34 and 2.3.24}\} \\
 & (I\!I_D \lhd b \rhd \top_D) ; P \\
= & \{\text{Definition 2.3.50}\} \\
 & b^\top ; P
\end{array}
$$

□

Law 2.3.73 $(\neg b)_\perp ; (P \lhd b \vee c \rhd Q) = (\neg b)_\perp ; (P \lhd c \rhd Q)$

Proof.

$$
\begin{array}{ll}
 & (\neg b)_\perp ; (P \lhd b \vee c \rhd Q) \\
= & \{\text{Definition 2.3.51 and laws 2.3.24, 2.3.10, 2.3.4}\} \\
 & (P \lhd b \vee c \rhd Q) \lhd \neg b \rhd \perp_D \\
= & \{\text{Law 2.3.28}\} \\
 & P \lhd (b \vee c) \wedge \neg b \rhd (Q \lhd \neg b \rhd \perp_D) \\
= & \{\text{Propositional calculus}\} \\
 & P \lhd c \wedge \neg b \rhd (Q \lhd \neg b \rhd \perp_D) \\
= & \{\text{Law 2.3.28}\} \\
 & (P \lhd c \rhd Q) \lhd \neg b \rhd \perp_D \\
= & \{\text{Laws 2.3.24, 2.3.10 and 2.3.4, then theorem 2.3.53}\} \\
 & (\neg b)_\perp ; (P \lhd c \rhd Q)
\end{array}
$$

□

Law 2.3.74 $b^\top ; (P \lhd b \vee c \rhd Q) = b^\top ; P$

Proof. Similar to the proof of law 2.3.73. □

Law 2.3.75 $(\neg b)^\top ; (P \lhd b \rhd Q) = (\neg b)^\top ; Q$

Proof. Direct from laws 2.3.23 and 2.3.72. □

Law 2.3.76 $b_\perp ; (P \lhd c \rhd Q) = (b_\perp ; P) \lhd c \rhd (b_\perp ; Q)$

Proof.

$$
\begin{array}{l}
\quad b_\perp ; (P \lhd c \rhd Q) \\
= \{\text{Definition 2.3.51}\} \\
\quad (I\!I_D \lhd b \rhd \perp_D) ; (P \lhd c \rhd Q) \\
= \{\text{Laws 2.3.24, 2.3.10, 2.3.4}\} \\
\quad (P \lhd c \rhd Q) \lhd b \rhd \perp_D \\
= \{\text{Law 2.3.30}\} \\
\quad (P \lhd b \rhd \perp_D) \lhd c \rhd (Q \lhd b \rhd \perp_D) \\
= \{\text{Laws 2.3.10, 2.3.4}\} \\
\quad (I\!I_D ; P \lhd b \rhd (\perp_D ; P)) \lhd c \rhd (I\!I_D ; Q \lhd b \rhd (\perp_D ; P)) \\
= \{\text{Law 2.3.24 and definition 2.3.51}\} \\
\quad (b_\perp ; P) \lhd c \rhd (b_\perp ; Q)
\end{array}
$$

□

Law 2.3.77 $b^\top ; (P \lhd c \rhd Q) = (b^\top ; P) \lhd c \rhd (b^\top ; Q)$

Proof.

Similar to the proof of 2.3.76, but using that $\top_D$ is also a left zero for sequential composition (law 2.3.34). □

Law 2.3.78 Provided that $(b \wedge c = false)$ we have that:

$$b^\top ; (P \lhd c \rhd Q) = b^\top ; Q$$

Proof.

$$
\begin{array}{l}
\quad b^\top ; (P \lhd c \rhd Q) \\
= \{\text{Definition 2.3.50}\} \\
\quad (I\!I_D \lhd b \rhd \top_D) ; (P \lhd c \rhd Q) \\
= \{\text{Laws 2.3.24, 2.3.10, 2.3.34}\} \\
\quad (P \lhd c \rhd Q) \lhd b \rhd \top_D \\
= \{\text{Law 2.3.28}\} \\
\quad P \lhd b \wedge c \rhd (Q \lhd b \rhd \top_D) \\
= \{\text{Proviso } (b \wedge c = false) \text{ and law 2.3.21}\} \\
\quad Q \lhd b \rhd \top_D
\end{array}
$$

= {Laws 2.3.10, 2.3.34 and 2.3.24}

$(I\!I_D \lhd b \rhd \top_D); Q$

= {Definition 2.3.50}

$b^\top; Q$ □

Law 2.3.79 $\mathbf{var}\ x; (x = e)^\top = \mathbf{var}\ x; x := e$

Proof.

$\mathbf{var}\ x; (x = e)^\top$

= {Definitions 2.3.35, 2.3.50, theorem 2.3.19 and propositional calculus}

$(\mathbf{true} \vdash \mathbf{var}\ x; (x = e)^\top)$

= {Definitions 2.2.18, 2.3.50, 2.2.13 and 2.2.1}

$(\mathbf{true} \vdash \exists x \bullet (x = e \land I\!I) \lor ((\neg x = e) \land \mathbf{false}))$

= {Propositional calculus, definition 2.2.7}

$(\mathbf{true} \vdash \exists x \bullet x = e \land x' = x)$

= {One-point rule}

$(\mathbf{true} \vdash x' = e)$

= {Spurious quantifier}

$(\mathbf{true} \vdash \exists x \bullet x' = e)$

= {Definitions 2.2.18 and 2.2.6}

$(\mathbf{true} \vdash \mathbf{var}\ x; x := e)$

= {Definitions 2.3.35, 2.3.50, theorem 2.3.19 and propositional calculus}

$\mathbf{var}\ x; x := e$ □

Law 2.3.85 $b^\top; b * P = b^\top; P; (b * P)$

Proof.

$b^\top; b * P$

= {Definition 2.3.82, law 2.3.80}

$b^\top; (P; b * P) \lhd b \rhd I\!I_D$

= {Law 2.3.72}

$b^\top; P; (b * P)$ □

Law 2.3.99 Provided that x is free in P and Q does not mention x we have that:

$\mathbf{var}\ x; (P \parallel Q) = (\mathbf{var}\ x; P) \parallel Q$

Proof.

$$
\begin{aligned}
& \mathbf{var}\ x; (P \parallel Q) \\
= & \ \{\text{Definitions 2.3.35 and 2.3.89}\} \\
& (\mathbf{true} \vdash \mathbf{var}\ x); (P_1 \wedge Q_1 \vdash P_2 \wedge Q_2) \\
= & \ \{\text{Theorem 2.3.19, definition 2.2.18}\} \\
& (\neg(\exists x \bullet \neg(P_1 \wedge Q_1)) \vdash \exists x \bullet (P_2 \wedge Q_2)) \\
= & \ \{\text{Assumption } (Q \text{ does not mention } x), \text{ propositional calculus}\} \\
& (\neg((\neg Q_1) \vee (\exists x \bullet \neg P_1)) \vdash (\exists x \bullet P_2) \wedge Q_2) \\
= & \ \{\text{Propositional calculus}\} \\
& (\neg(\exists x \bullet \neg P_1) \wedge Q_1 \vdash (\exists x \bullet P_2) \wedge Q_2) \\
= & \ \{\text{Definitions 2.3.89 and 2.2.18, propositional calculus}\} \\
& (\mathbf{true} \wedge \neg(\mathbf{var}\ x; \neg P_1) \vdash (\mathbf{var}\ x; P_2)) \parallel Q \\
= & \ \{\text{Theorem 2.3.19 and definitions 2.3.35}\} \\
& (\mathbf{var}\ x; P) \parallel Q
\end{aligned}
$$

□

Law 2.3.100 Provided that x is free in P, Q does not mention x and P and Q are **H3**, we have that:

$$(P \parallel Q); \mathbf{end}\ x = (P; \mathbf{end}\ x) \parallel Q$$

Proof.

$$
\begin{aligned}
& (P \parallel Q); \mathbf{end}\ x \\
= & \ \{\text{Definitions 2.3.89 and 2.3.36}\} \\
& (P_1 \wedge Q_1 \vdash P_2 \wedge Q_2); (\mathbf{true} \vdash \mathbf{end}\ x) \\
= & \ \{\text{Theorem 2.3.19 and definition 2.2.19}\} \\
& (\neg(\neg(P_1 \wedge Q_1); \mathbf{true}) \wedge \neg(P_2 \wedge Q_2; \neg\mathbf{true}) \vdash \exists x' \bullet P_2 \wedge Q_2) \\
= & \ \{\text{Propositional calculus, assumption and quantifier contract scope}\} \\
& (\neg(\neg(P_1 \wedge Q_1); \mathbf{true}) \vdash (\exists x' \bullet P_2) \wedge Q_2) \\
= & \ \{\text{Assumption } P \text{ and } Q \text{ are } \mathbf{H3} \rightarrow P \parallel Q \text{ is also } \mathbf{H3}\} \\
& (P_1 \wedge Q_1 \vdash (\exists x' \bullet P_2) \wedge Q_2) \\
= & \ \{\text{Assumption } P \text{ is } \mathbf{H3}\} \\
& (\neg(\neg P_1; \mathbf{true}) \wedge Q_1 \vdash (\exists x' \bullet P_2) \wedge Q_2) \\
= & \ \{\text{Propositional calculus, definitions 2.3.89 and 2.3.36}\} \\
& (\neg(\neg P_1; \mathbf{true}) \wedge \neg(P_2; \neg\mathbf{true}) \vdash \exists x' \bullet P_2) \parallel Q \\
= & \ \{\text{Theorem 2.3.19 and definition 2.2.19}\}
\end{aligned}
$$

$(P; \textbf{end}\ x) \parallel Q$ □

Law 2.3.102 Provided that x and y are different variables we have that:

$$U0(x, y) = U0(y, x)$$

Proof.

$U0(x, y)$
= {Definition 2.3.101}
$\textbf{var}\ 0.x, 0.y := x, y; \textbf{end}\ x, y$
= {Laws 2.3.44, 2.3.45 and 2.3.16}
$\textbf{var}\ 0.y, 0.x := y, x; \textbf{end}\ y, x$
= {Definition 2.3.101}
$U0(y, x)$ □

Law 2.3.103 Provided that x and y are different variables we have that:

$$U0(x, y) = U0(x); U0(y)$$

Proof.

$U0(x, y)$
= {Definition 2.3.101}
$\textbf{var}\ 0.x, 0.y := x, y; \textbf{end}\ x, y$
= {Laws 2.3.42, 2.3.43}
$\textbf{var}\ 0.x; \textbf{var}\ 0.y; (0.x, 0.y := x, y); \textbf{end}\ x; \textbf{end}\ y$
= {Law 2.3.11}
$\textbf{var}\ 0.x; \textbf{var}\ 0.y; (0.x := x)_{+0.y}; (0.y := y)_{+0.x}; \textbf{end}\ x; \textbf{end}\ y$
= {Laws 2.3.48 and 2.3.47}
$\textbf{var}\ 0.x; (0.x := x); \textbf{end}\ x; \textbf{var}\ 0.y; (0.y := y); \textbf{end}\ y$
= {Definition 2.3.101}
$U0(x); U0(y)$ □

Theorem 2.3.122 Selection function – M equivalence

$$M(m, 0.m, 1.m, m') \sqsubseteq m := \texttt{SELECT}\,(0.m, 1.m, m)$$

Proof.

$$
\begin{aligned}
& M(m, 0.m, 1.m, m') \\
= & \{\text{Definition 2.3.114, laws 2.2.11 and 2.3.22}\} \\
& m' = ((m \lhd m = 1.m \rhd 1.m) \lhd m = 0.m \rhd 0.m) \\
= & \{\text{Definitions 2.3.121 and 2.2.6}\} \\
& m \overset{:=}{_{\text{snc}}} \texttt{SELECT}\,(0.m, 1.m, m)
\end{aligned}
$$

□

Law 2.3.106 $P \parallel_M Q = Q \parallel_M P$

Law 2.3.107 $P \parallel (Q \parallel_M R) = (P \parallel_M Q) \parallel_M R$

Law 2.3.108 $(I\!I_A \parallel_M P) = P_{+A}$

Law 2.3.109 $\mathbf{true} \parallel_M P = \mathbf{true}$

Law 2.3.110 $(P \lhd b \rhd Q) \parallel_M R = ((P \parallel R) \lhd b \rhd (Q \parallel R))$

Law 2.3.111 For any descending chain $S = \{S_n \mid n \in \mathcal{N}\}$ we have that:

$$(\bigsqcap S) \parallel_M R = \bigsqcap_n (S_n \parallel_M R)$$

Law 2.3.112 Provided that $x := e$ does not mention m we have that:

$$(x := e; P) \parallel_M Q = (x := e); (P \parallel_M Q)$$

Proof.

The proofs of laws 2.3.106 to 2.3.112 follow from the merge operator satisfying laws 2.3.115 to 2.3.116 and [Hoare and He 1998, theorem 7.2.10] □

Law 2.3.115 $(0.m, 1.m := 1.m, 0.m); M = M$

We begin by showing M can be re-written in a more convenient fashion:

Lemma A.0.1. *Merge function (equivalent formulation)*

$$
\begin{aligned}
M(m, 0.m, 1.m, m') =_{df} \; & (m = 0.m \wedge m = 1.m \wedge m' = m) \vee \\
& (m = 0.m \wedge m \neq 1.m \wedge m' = 1.m) \vee \\
& (m \neq 0.m \wedge m = 1.m \wedge m' = 0.m) \vee \\
& (m \neq 0.m \wedge m \neq 1.m \wedge m' = (1.m \sqcap 0.m))
\end{aligned}
$$

Proof.

Straightforward from definitions 2.3.114, 2.2.1 and 2.2.6. □

Based on this result, the proof of law 2.3.115 goes as follows:

Proof.

$$(0.m, 1.m := 1.m, 0.m); M(m, 0.m, 1.m, m')$$

= {Law 2.3.11 and lemma A.0.1}

$(m = 1.m \wedge m = 0.m \wedge m' = m) \vee (m = 1.m \wedge m \neq 0.m \wedge m' = 0.m) \vee$

$(m \neq 1.m \wedge m = 0.m \wedge m' = 1.m) \vee (m \neq 1.m \wedge m \neq 0.m \wedge m' = (0.m \sqcap 1.m))$

= {Propositional calculus ($\wedge$-comm, $\vee$-comm) and $\sqcap$-comm}

$(m = 0.m \wedge m = 1.m \wedge m' = m) \vee (m = 0.m \wedge m \neq 1.m \wedge m' = 1.m) \vee$

$(m \neq 0.m \wedge m = 1.m \wedge m' = 0.m) \vee (m \neq 0.m \wedge m \neq 1.m \wedge m' = (1.m \sqcap 0.m))$

= {Lemma A.0.1}

$M(m, 0.m, 1.m, m')$ □

Law 2.3.116 $(0.m, 1.m := m, m); M = \mathit{II}$

Proof.

Straightforward from definition 2.3.114, law 2.3.11 and predicate calculus. □

Law 2.3.117 $(0.m, 1.m, 2.m := 1.m, 2.m, 0.m); M3 = M3$

Proof.

We begin by reducing $M3$ to a more amenable form:

$M3$

= {Definition of $M3$, lemma A.0.1}

$$\exists x \bullet \left(\begin{array}{l}
(m = 0.m \land m = 1.m \land x = m \land m = x \land m = 2.m \land m' = m) \lor \\
(m = 0.m \land m = 1.m \land x = m \land m = x \land m \neq 2.m \land m' = 2.m) \lor \\
(m = 0.m \land m = 1.m \land x = m \land m \neq x \land m = 2.m \land m' = x) \lor \\
(m = 0.m \land m = 1.m \land x = m \land m \neq x \land m \neq 2.m \land m' = (2.m \sqcap x)) \lor \\
(m = 0.m \land m \neq 1.m \land x = 1.m \land m = x \land m = 2.m \land m' = m) \lor \\
(m = 0.m \land m \neq 1.m \land x = 1.m \land m = x \land m \neq 2.m \land m' = 2.m) \lor \\
(m = 0.m \land m \neq 1.m \land x = 1.m \land m \neq x \land m = 2.m \land m' = x) \lor \\
(m = 0.m \land m \neq 1.m \land x = 1.m \land m \neq x \land m \neq 2.m \land m' = (2.m \sqcap x)) \lor \\
(m \neq 0.m \land m = 1.m \land x = 0.m \land m = x \land m = 2.m \land m' = m) \lor \\
(m \neq 0.m \land m = 1.m \land x = 0.m \land m = x \land m \neq 2.m \land m' = 2.m) \lor \\
(m \neq 0.m \land m = 1.m \land x = 0.m \land m \neq x \land m = 2.m \land m' = x) \lor \\
(m \neq 0.m \land m = 1.m \land x = 0.m \land m \neq x \land m \neq 2.m \land m' = (2.m \sqcap x)) \lor \\
(m \neq 0.m \land m \neq 1.m \land x = (1.m \sqcap 0.m) \land m = x \land m = 2.m \land m' = m) \lor \\
(m \neq 0.m \land m \neq 1.m \land x = (1.m \sqcap 0.m) \land m = x \land m \neq 2.m \land m' = 2.m) \lor \\
(m \neq 0.m \land m \neq 1.m \land x = (1.m \sqcap 0.m) \land m \neq x \land m = 2.m \land m' = x) \lor \\
(m \neq 0.m \land m \neq 1.m \land x = (1.m \sqcap 0.m) \land m \neq x \land m \neq 2.m \land m' = (2.m \sqcap x))
\end{array}\right)$$

= {$\exists$-assoc, one-point rule, propositional calculus and equality substitution}

$(m = 0.m \land m = 1.m \land m = 2.m \land m' = m) \lor$

$(m = 0.m \land m = 1.m \land m \neq 2.m \land m' = 2.m) \lor$

$(m = 0.m \land m \neq 1.m \land m = 2.m \land m' = 1.m) \lor$

$(m = 0.m \land m \neq 1.m \land m \neq 2.m \land m' = (1.m \sqcap 2.m)) \lor$

$(m \neq 0.m \land m = 1.m \land m = 2.m \land m' = 0.m) \lor$

$(m \neq 0.m \land m = 1.m \land m \neq 2.m \land m' = (0.m \sqcap 2.m)) \lor$

$(m \neq 0.m \land m \neq 1.m \land m = 2.m \land m' = (0.m \sqcap 1.m)) \lor$

$(m \neq 0.m \land m \neq 1.m \land m \neq 2.m \land m' = (0.m \sqcap 1.m \sqcap 2.m))$

The proof the associative property follows:

$(0.m, 1.m, 2.m := 1.m, 2.m, 0.m); M3(m, 0.m, 1.m, 2.m, m')$

= {Law 2.3.11 and $M3$ expansion above}

$(m = 1.m \land m = 2.m \land m = 0.m \land m' = m) \lor$

$(m = 1.m \land m = 2.m \land m \neq 0.m \land m' = 0.m) \lor$

$(m = 1.m \land m \neq 2.m \land m = 0.m \land m' = 2.m) \lor$

$(m = 1.m \land m \neq 2.m \land m \neq 0.m \land m' = (2.m \sqcap 0.m)) \lor$

$(m \neq 1.m \land m = 2.m \land m = 0.m \land m' = 1.m) \lor$

$(m \neq 1.m \wedge m = 2.m \wedge m \neq 0.m \wedge m' = (1.m \sqcap 0.m)) \vee$

$(m \neq 1.m \wedge m \neq 2.m \wedge m = 0.m \wedge m' = (1.m \sqcap 2.m)) \vee$

$(m \neq 1.m \wedge m \neq 2.m \wedge m \neq 0.m \wedge m' = (1.m \sqcap 2.m \sqcap 0.m))$

= {Propositional calculus ($\wedge$-comm, $\vee$-comm) and $\sqcap$-comm}

$(m = 0.m \wedge m = 1.m \wedge m = 2.m \wedge m' = m) \vee$

$(m = 0.m \wedge m = 1.m \wedge m \neq 2.m \wedge m' = 2.m) \vee$

$(m = 0.m \wedge m \neq 1.m \wedge m = 2.m \wedge m' = 1.m) \vee$

$(m = 0.m \wedge m \neq 1.m \wedge m \neq 2.m \wedge m' = (1.m \sqcap 2.m)) \vee$

$(m \neq 0.m \wedge m = 1.m \wedge m = 2.m \wedge m' = 0.m) \vee$

$(m \neq 0.m \wedge m = 1.m \wedge m \neq 2.m \wedge m' = (0.m \sqcap 2.m)) \vee$

$(m \neq 0.m \wedge m \neq 1.m \wedge m = 2.m \wedge m' = (0.m \sqcap 1.m)) \vee$

$(m \neq 0.m \wedge m \neq 1.m \wedge m \neq 2.m \wedge m' = (0.m \sqcap 1.m \sqcap 2.m))$

= {$M3$ expansion}

$M3(m, 0.m, 1.m, 2.m, m')$ □

Theorem 2.3.120 *M preserves **H1** to **H4**.*

Proof.

The above claim can be interpreted as: provided that $P = (P_1 \vdash P_2)$ is **H1** to **H4** then $P_{+m}; M_{+0.m,1.m}$ is also **H1** to **H4**. From this observation we have that:

$P_{+m}; M_{+0.m,1.m}$

= {Assumption $P = (P_1 \vdash P_2)$, definition 2.3.46}

$(P_1 \vdash P_2 \wedge m' = m); M \wedge 0.m', 1.m' = 0.m, 1.m$

= {Definition 2.3.1, propositional calculus, disjunctivity of sequential composition}

$(\neg ok; M \wedge 0.m', 1.m' = 0.m, 1.m) \vee$

$((\neg P_1 \wedge m' = m); M \wedge 0.m', 1.m' = 0.m, 1.m) \vee$

$((ok' \wedge P_2 \wedge m' = m); M \wedge 0.m', 1.m' = 0.m, 1.m)$

= {Definition 2.2.3}

$(\exists ok_0, 0.m_0, 1.m_0, m_0 \bullet \neg ok \wedge M[m_0/m] \wedge 0.m', 1.m' = 0.m_0, 1.m_0) \vee$

$(\exists ok_0, 0.m_0, 1.m_0, m_0 \bullet \neg P_1[0.m_0, 1.m_0/0.m', 1.m'] \wedge m_0 = m \wedge$

$\quad M[m_0/m] \wedge 0.m', 1.m' = 0.m_0, 1.m_0) \vee$

$(\exists ok_0, 0.m_0, 1.m_0, m_0 \bullet (ok_0 \wedge P_2[m_0, 0.m_0, 1.m_0/m', 0.m', 1.m'] \wedge m_0 = m \wedge$

$\quad M[m_0/m] \wedge 0.m', 1.m' = 0.m_0, 1.m_0))$

= {One-point rule, P is **H3**}

$$
\begin{aligned}
& \neg ok \lor \neg P_1 \lor (\exists ok_0 \bullet (ok_0 \land P_2 \land M(m, 0.m', 1.m', m'))) \\
= & \{ok_0 = ok'\} \\
& \neg ok \lor \neg P_1 \lor (ok' \land P_2 \land M(m, 0.m', 1.m', m')) \\
= & \{\text{Propositional calculus}\} \\
& (ok \land P_1 \Rightarrow (ok' \land P_2 \land M(m, 0.m', 1.m', m')))
\end{aligned}
$$

From this point, it is easy to see that the resulting expression is **H1** and **H2**. Also, as P is **H3**, then P_1 does not mention dashed variables. Then, the resulting expression is also **H3**. The result is also **H4** because the resulting expression is only adding a restriction on m'. The result of M is to select either $0.m$ or $1.m$. As P_2 is **H4**, we know a witness for m' exists. □

Appendix B

Proofs from Chapter 3

Theorem 3.3.6 Sequential composition of **S**-healthy designs

$$\mathbf{S}(P_1 \vdash P_2); \mathbf{S}(Q_1 \vdash Q_2) = \mathbf{S}(\neg(\mathbf{S}(\neg P_1); \mathbf{S}(\mathbf{true})) \wedge \neg(\mathbf{S}(P_2); \mathbf{S}(\neg Q_1)) \vdash \mathbf{S}(P_2); \mathbf{S}(Q_2))$$

Proof.

$$\begin{array}{ll}
& \mathbf{S}(P_1 \vdash P_2); \mathbf{S}(Q_1 \vdash Q_2) \\
= & \{\text{Definition 2.3.1 and propositional calculus}\} \\
& \mathbf{S}(\neg ok) \vee \mathbf{S}(\neg P_1) \vee \mathbf{S}(ok' \wedge P_2); \mathbf{S}(Q_1 \vdash Q_2) \\
= & \{\text{Disjuntivity of sequential composition}\} \\
& \mathbf{S}(\neg ok); \mathbf{S}(Q_1 \vdash Q_2) \vee \mathbf{S}(\neg P_1); \mathbf{S}(Q_1 \vdash Q_2) \vee \mathbf{S}(ok' \wedge P_2); \mathbf{S}(Q_1 \vdash Q_2) \\
= & \{\text{Definition 2.2.3}\} \\
& \exists ok_0, v_0 \bullet \mathbf{S}(\neg ok)[ok_0, v_0/ok', v']; \mathbf{S}(Q_1 \vdash Q_2)[ok_0, v_0/ok, v] \vee \\
& \exists ok_0, v_0 \bullet \mathbf{S}(\neg P_1)[ok_0, v_0/ok', v']; \mathbf{S}(Q_1 \vdash Q_2)[ok_0, v_0/ok, v] \vee \\
& \exists ok_0, v_0 \bullet \mathbf{S}(ok' \wedge P_2)[ok_0, v_0/ok', v']; \mathbf{S}(Q_1 \vdash Q_2)[ok_0, v_0/ok, v] \\
= & \{\mathbf{S} \text{ does not mention } ok \text{ and } ok', \text{quantifier contract scope, predicate calculus}\} \\
& \exists v_0 \bullet \mathbf{S}(\neg ok)[v_0/v'] \wedge \mathbf{S}(\exists ok_0 \bullet Q_1 \vdash Q_2)[v_0/v] \vee \\
& \exists v_0 \bullet \mathbf{S}(\neg P_1)[v_0/v'] \wedge \mathbf{S}(\exists ok_0 \bullet Q_1 \vdash Q_2)[v_0/v] \vee \\
& \exists v_0 \bullet \mathbf{S}(P_2)[v_0/v'] \wedge \mathbf{S}(Q_1 \vdash Q_2)[\mathbf{true}, v_0/ok, v] \\
= & \{\text{For any design } P \text{ we have } (\exists ok \bullet P) = \mathbf{true}, \text{definition 2.3.1 and propositional calculus}\} \\
& \neg ok \wedge \exists v_0 \bullet \mathbf{S}(\mathbf{true})[v_0/v'] \wedge \mathbf{S}(\mathbf{true})[v_0/v] \vee \\
& \exists v_0 \bullet \mathbf{S}(\neg P_1)[v_0/v'] \wedge \mathbf{S}(\mathbf{true})[v_0/v] \vee \\
& \exists v_0 \bullet \mathbf{S}(P_2)[v_0/v']; \mathbf{S}(\neg ok)[\mathbf{true}, v_0/ok, v] \vee \mathbf{S}(\neg Q_1)[v_0/v] \vee \mathbf{S}(ok' \wedge Q_2)[v_0/v] \\
= & \{\text{Theorem 3.2.10, propositional calculus, disjunctivity of sequential composition}\} \\
& \mathbf{S}(\neg ok) \vee (\mathbf{S}(\neg P_1); \mathbf{S}(\mathbf{true})) \vee \\
& \exists v_0 \bullet (\mathbf{S}(P_2)[v_0/v']; \mathbf{S}(\neg Q_1)[v_0/v]) \vee (\mathbf{S}(P_2)[v_0/v']; \mathbf{S}(ok' \wedge Q_2)[v_0/v])
\end{array}$$

= {∃-associtivity, definition 2.2.3}

$\mathbf{S}(\neg ok) \vee (\mathbf{S}(\neg P_1); \mathbf{S}(\mathbf{true})) \vee (\mathbf{S}(P_2); \mathbf{S}(\neg Q_1)) \vee \mathbf{S}(ok' \wedge (\mathbf{S}(P_2); \mathbf{S}(Q_2)))$

= {Theorem 3.2.13, propositional calculus}

$\mathbf{S}(ok \wedge \neg(\mathbf{S}(\neg P_1); \mathbf{S}(\mathbf{true})) \wedge \neg(\mathbf{S}(P_2); \mathbf{S}(\neg Q_1)) \Rightarrow ok' \wedge (\mathbf{S}(P_2); \mathbf{S}(Q_2)))$

= {Definition 2.3.1}

$\mathbf{S}(\neg(\mathbf{S}(\neg P_1); \mathbf{S}(\mathbf{true})) \wedge \neg(\mathbf{S}(P_2); \mathbf{S}(\neg Q_1)) \vdash \mathbf{S}(P_2); \mathbf{S}(Q_2))$ □

Theorem 3.2.8

$$I\!I \Rightarrow \mathbf{S}(\mathbf{true})$$

Proof.

Trivial from definition 2.2.7 and the fact that $x.out' = x.out \Rightarrow x.out \leq x.out'$. □

Theorem 3.2.9 $\mathbf{S}(I\!I) = I\!I$

Proof.

$\mathbf{S}(I\!I)$

= {Definition 3.2.2}

$I\!I \wedge \mathbf{S}$

= {Definitions 3.2.2, 2.2.7 and 3.2.1; propositional calculus}

$c', x', x.in', x.out', v' = c, x, x.in, x.out \wedge x.in' = x.in \wedge x.out \leq x.out' \wedge c \leq c'$

= {Propositional calculus}

$c', x', x.in', x.out', v' = c, x, x.in, x.out$

= {Definition 2.2.7}

$I\!I$ □

Theorem 3.2.10 **S** is transitive $\mathbf{S}(\mathbf{true}); \mathbf{S}(\mathbf{true}) = \mathbf{S}(\mathbf{true})$

Proof.

Straightforward from definitions 3.2.2, 3.2.1 and 2.2.3, and transitivity of sequence prefix. □

Theorem 3.2.11 **S** is monotonic

$$(P \sqsubseteq Q) \Rightarrow (\mathbf{S}(P) \sqsubseteq \mathbf{S}(Q))$$

provided P and Q are designs in the sense of the previous chapter.

Proof.

$\mathbf{S}(P) \sqsubseteq \mathbf{S}(Q)$

$\equiv$ {Definition 3.2.2}

$P \wedge \mathbf{S} \sqsubseteq Q \wedge \mathbf{S}$

$\equiv$ {Definition of refinement}

$[Q \wedge \mathbf{S} \Rightarrow P \wedge \mathbf{S}]$

$\equiv$ {Predicate calculus}

$[Q \wedge \mathbf{S} \Rightarrow P]$

$\Leftarrow$ {Predicate calculus}

$[Q \Rightarrow P]$

$\equiv$ {Definition of refinement}

$P \sqsubseteq Q$ □

Theorem 3.2.12 **S** distributes over conjunction

$$\mathbf{S}(P \wedge Q) = \mathbf{S}(P) \wedge \mathbf{S}(Q)$$

Proof.

$\mathbf{S}(P \wedge Q)$

$=$ {Definition 3.2.2}

$(P \wedge Q) \wedge \mathbf{S}$

$=$ {Propositional calculus}

$(P \wedge \mathbf{S}) \wedge (Q \wedge \mathbf{S})$

$=$ {Definition 3.2.2}

$\mathbf{S}(P) \wedge \mathbf{S}(Q)$ □

Theorem 3.2.13 **S** distributes over disjunction

$$\mathbf{S}(P \vee Q) = \mathbf{S}(P) \vee \mathbf{S}(Q)$$

Proof.

$\mathbf{S}(P) \vee \mathbf{S}(Q)$

$=$ {Definition 3.2.2}

$(P \wedge \mathbf{S}) \vee (Q \wedge \mathbf{S})$

$=$ {Propositional calculus}

$(P \vee Q) \wedge \mathbf{S}$

$=$ {Definition 3.2.2}

$\mathbf{S}(P \vee Q)$ □

Theorem 3.2.14 **S** extends over conjunction

$$P \wedge \mathbf{S}(Q) = \mathbf{S}(P \wedge Q)$$

Proof.

$$\begin{aligned}
& P \wedge \mathbf{S}(Q) \\
= & \{\text{Definition 3.2.2}\} \\
& P \wedge Q \wedge \mathbf{S} \\
= & \{\wedge\text{-associative, definition 3.2.2}\} \\
& \mathbf{S}(P \wedge Q)
\end{aligned}$$

□

Theorem 3.2.16 Synchronous assumption and precondition
A **S** healthy design $\mathbf{S}(P_1 \vdash P_2)$ is **SH3 iff** its precondition does not mention dashed variables other than the observations c, $x.out$ and $x.in$.

Proof.

$$\begin{aligned}
& \mathbf{S}(P_1 \vdash P_2) = \mathbf{S}(P_1 \vdash P_2); \mathbb{I} \\
= & \{\text{Theorem 3.3.6, predicate and propositional calculus}\} \\
& \mathbf{S}(P_1 \vdash P_2) = \mathbf{S}(\neg(\mathbf{S}(\neg P_1); \mathbf{S}(\mathbf{true})) \vdash \mathbf{S}(P_2)) \\
= & \{\text{Propositional calculus}\} \\
& \mathbf{S}(P_1 \vdash P_2) = \mathbf{S}(\neg(\mathbf{S}(\neg P_1); \mathbf{S}(\mathbf{true})) \vdash \mathbf{S}(P_2)) \\
= & \{\text{Definition 2.3.1 and propositional calculus}\} \\
& \mathbf{S}(\neg\mathbf{S}(\neg P_1) \vdash \mathbf{S}(P_2)) = \mathbf{S}(\neg(\mathbf{S}(\neg P_1); \mathbf{S}(\mathbf{true})) \vdash \mathbf{S}(P_2)) \\
= & \{\text{Design equality, propositional calculus}\} \\
& \mathbf{S}(\neg P_1) = \mathbf{S}(\neg P_1); \mathbf{S}(\mathbf{true}) \\
= & \{\text{Predicate calculus}\} \\
& \mathbf{S}(P_1) = \mathbf{S}(P_1); \mathbf{S}(\mathbf{true})
\end{aligned}$$

□

Theorem 3.2.17 **SH3** $\circ$ **S** sequential composition closedness

$$P; Q = \mathbf{SH3}(P; Q)$$

provided P and Q are **SH3**-healthy.

Proof.

$$\begin{aligned}
& P; Q \\
= & \{\text{Assumption } (Q \text{ is } \mathbf{SH3} \circ \mathbf{S})\} \\
& P; (Q; \mathbb{I})
\end{aligned}$$

= {Law 2.2.4}

$(P; Q); \Pi$

= {Definition 3.2.15}

$\mathbf{SH3}(P; Q)$ □

Theorem 3.2.18 $\mathbf{SH3} \circ \mathbf{S}$ conditional closedness

$$P \lhd b \rhd Q = \mathbf{SH3}(P \lhd b \rhd Q)$$

provided P and Q are **SH3**-healthy.

Proof.

$P \lhd b \rhd Q$

= {Assumption (P and Q are $\mathbf{SH3} \circ \mathbf{S}$)}

$(P; \Pi) \lhd b \rhd (Q; \Pi)$

= {Law 3.3.33}

$(P \lhd b \rhd Q); \Pi$

= {Definition 3.2.15}

$\mathbf{SH3}(P \lhd b \rhd Q)$ □

Law 3.3.4 $\top \sqsupseteq \mathbf{S}(P)$

Proof.

true

≡ {Law 2.2.15}

false $\sqsupseteq P$

⇒ {Theorem 3.2.11}

$\mathbf{S}(\mathbf{false}) \sqsupseteq \mathbf{S}(P)$

≡ {Definition 3.3.1}

$\top \sqsupseteq \mathbf{S}(P)$ □

Law 3.3.5 $\bot \sqsubseteq \mathbf{S}(P)$

Proof.

true

≡ {Law 2.2.14}

true $\sqsubseteq P$

⇒ {Theorem 3.2.11}

$$
\begin{array}{ll}
& \mathbf{S}(\mathbf{true}) \sqsubseteq \mathbf{S}(P) \\
\equiv & \{\text{Definition 3.3.2}\} \\
& \bot \sqsubseteq \mathbf{S}(P)
\end{array}
$$

□

Law 3.3.7 $\top ; \mathbf{S}(P) = \top$

Proof.

$$
\begin{array}{ll}
& \top ; \mathbf{S}(P) \\
= & \{\text{Definitions 3.3.1 and 3.2.2, theorem 2.3.5 and propositional calculus}\} \\
& \neg ok \wedge \mathbf{S}; P \wedge \mathbf{S} \\
= & \{\text{Definition 2.2.3, } \mathbf{S} \text{ does not mention } ok \text{ or } ok'\} \\
& \exists ok_0, v_0 \bullet \neg ok \wedge \mathbf{S}[v_0/v'] \wedge P[ok_0, v_0/ok, v] \wedge \mathbf{S}[v_0/v] \\
= & \{\text{Propositional calculus}\} \\
& \neg ok \wedge \exists v_0 \bullet \mathbf{S}[v_0/v'] \wedge \mathbf{S}[v_0/v] \wedge (\exists ok_0 \bullet P[ok_0, v_0/ok, v]) \\
= & \{\text{For any design } P \text{ we have } (\exists ok \bullet P) = \mathbf{true}\text{, definition 2.3.1 and propositional calculus}\} \\
& \neg ok \wedge \exists v_0 \bullet \mathbf{S}[v_0/v'] \wedge \mathbf{S}[v_0/v] \\
= & \{\text{Definition 2.2.3 and theorem 3.2.10}\} \\
& \neg ok \wedge \mathbf{S} \\
= & \{\text{Theorem 2.3.5 then definitions 3.3.1 and 3.2.2}\} \\
& \top
\end{array}
$$

□

Law 3.3.8 $\bot ; \mathbf{S}(P) = \bot$

We begin by proving the following lemma:

Lemma B.0.2. *For a design P we have:*

$$\neg ok \wedge P = \neg ok$$

Proof.

$$
\begin{array}{ll}
& \neg ok \wedge P \\
= & \{\text{P is a design}\} \\
& \neg ok \wedge (ok \Rightarrow P) \\
= & \{\text{Propositional calculus}\} \\
& \neg ok \wedge (\mathbf{false} \Rightarrow P) \\
= & \{\text{Propositional calculus}\} \\
& \neg ok
\end{array}
$$

□

We can now prove law 3.3.8:

Proof.

$$
\begin{array}{ll}
& \bot ; \mathbf{S}(P) \\
= & \{\text{Definitions 3.3.2 and 3.2.2}\} \\
& \mathbf{true} \wedge \mathbf{S} ; P \wedge \mathbf{S} \\
= & \{\text{Propositional calculus}\} \\
& \mathbf{S} ; P \wedge \mathbf{S} \\
= & \{\text{Relational calculus, } ok, ok' \text{ not free in } \mathbf{S}\} \\
& \mathbf{S} ; (\exists ok \bullet P) \wedge \mathbf{S} \\
= & \{\text{Lemma B.0.2}\} \\
& \mathbf{S} ; \mathbf{true} \wedge \mathbf{S} \\
= & \{\text{Propositional calculus}\} \\
& \mathbf{S} ; \mathbf{S} \\
= & \{\text{Theorem 3.2.10 and propositional calculus}\} \\
& \mathbf{S}(\mathbf{true}) \\
= & \{\text{Definition 3.3.2}\} \\
& \bot
\end{array}
$$

□

Law 3.3.9 $\Pi ; \mathbf{S}(P_1 \vdash P_2) = \mathbf{S}(P_1 \vdash P_2)$

Proof.

$$
\begin{array}{ll}
& \Pi ; \mathbf{S}(P_1 \vdash P_2) \\
= & \{\text{Definition 3.2.7, theorem 3.3.6, propositional calculus}\} \\
& \mathbf{S}(\neg(\mathbf{S}(I\!I) ; \mathbf{S}(\neg P_1)) \vdash \mathbf{S}(I\!I) ; \mathbf{S}(P_2)) \\
= & \{\text{Theorem 3.2.9, law 2.2.8}\} \\
& \mathbf{S}(\neg \mathbf{S}(\neg P_1) \vdash \mathbf{S}(P_2)) \\
= & \{\text{Definition 2.3.1 and propositional calculus}\} \\
& \mathbf{S}(\neg ok \vee \mathbf{S}(\neg P_1) \vee (ok' \wedge \mathbf{S}(P_2)) \\
= & \{\text{Theorems 3.2.14 and 3.2.13, } \mathbf{S}\text{-idempotent}\} \\
& \mathbf{S}(\neg ok \vee \neg P_1 \vee (ok' \wedge P_2)) \\
= & \{\text{Propositional calculus, definition 2.3.1}\} \\
& \mathbf{S}(P_1 \vdash P_2)
\end{array}
$$

□

Law 3.3.10 Provided P is **SH3** we have that

$$\mathbf{S}(P) ; \Pi = \mathbf{S}(P)$$

Proof.

Straightforward from algebraic characterisation of **SH3**. □

Law 3.3.11 $(\bigsqcup S); P = \bigsqcup_i (S_i; P)$

Proof.

Straightforward from the fact that **S** is monotonic and it is conjunctive (hence it distributes over generalised disjunction). The distributivity of P is ensured by law 2.3.135. □

Law 3.3.12 Provided P is a finite normal form, we have:

$$P; (\bigsqcup S) = \bigsqcup_i (P; S_i)$$

Proof.

Straightforward from a similar argument to the one used in the proof of law 3.3.11. The distributivity of P is ensured by law 2.3.136. □

Theorem 3.3.14 Provided $x \notin \{c, x.in, x.out\}$ we have:

$$\mathbf{S}(x := e) = x := e$$

Proof.

Similar to the proof of theorem 3.2.9. □

Theorem 3.3.15 The assignment construct is **SH3**

Proof.

$x \stackrel{:=}{_{\mathrm{snc}}} e; \text{Ⅱ}$

= {Definitions 3.3.13 and 3.2.7}

$\mathbf{S}(\mathbf{true} \vdash x := e); \mathbf{S}(\mathbf{true} \vdash \text{Ⅱ})$

= {Theorem 3.3.6 and propositional calculus}

$\mathbf{S}(\mathbf{true} \vdash \mathbf{S}(x := e); \mathbf{S}(\text{Ⅱ}))$

= {Theorems 3.2.9 and 3.3.14, law 2.2.8}

$\mathbf{S}(\mathbf{true} \vdash x := e)$

= {Definition 3.3.13}

$x \stackrel{:=}{_{\mathrm{snc}}} e$ □

Law 3.3.16 $(x \stackrel{:=}{_{\mathrm{snc}}} e; y \stackrel{:=}{_{\mathrm{snc}}} f(x)) = y \stackrel{:=}{_{\mathrm{snc}}} f(e)$

Proof.

$x \stackrel{:=}{_{\mathrm{snc}}} e; y \stackrel{:=}{_{\mathrm{snc}}} f(x)$

= {Definition 3.3.13}

$$
\begin{aligned}
& \mathbf{S}(\mathbf{true} \vdash x := e); \mathbf{S}(\mathbf{true} \vdash y := f(x)) \\
= & \quad \{\text{Theorem 3.3.6 and propositional calculus}\} \\
& \mathbf{S}(\mathbf{true} \vdash \mathbf{S}(x := e); \mathbf{S}(y := f(x))) \\
= & \quad \{\text{Theorem 3.3.14 and law 2.3.11}\} \\
& \mathbf{S}(\mathbf{true} \vdash y := f(e)) \\
= & \quad \{\text{Definition 3.3.13}\} \\
& y \overset{:=}{_{\text{snc}}} f(e)
\end{aligned}
$$

□

Law 3.3.17 If $x \neq y$ and e_2 does not mention x we have:

$$(x \overset{:=}{_{\text{snc}}} e_1; y \overset{:=}{_{\text{snc}}} e_2) = (x, y \overset{:=}{_{\text{snc}}} e_1, e_2)$$

Proof.

$$
\begin{aligned}
& (x \overset{:=}{_{\text{snc}}} e_1; y \overset{:=}{_{\text{snc}}} e_2) \\
= & \quad \{\text{Definition 3.3.13}\} \\
& \mathbf{S}(\mathbf{true} \vdash x := e_1); \mathbf{S}(\mathbf{true} \vdash y := e_2) \\
= & \quad \{\text{Propositional and predicate calculus, theorem 3.3.14}\} \\
& \mathbf{S}(\mathbf{true} \vdash x := e_1; y := e_2) \\
= & \quad \{\text{Law 2.3.17 and definition 3.3.13}\} \\
& x, y \overset{:=}{_{\text{snc}}} e_1, e_2
\end{aligned}
$$

□

Law 3.3.18 $(x, y \overset{:=}{_{\text{snc}}} e_1, e_2) = (y, x \overset{:=}{_{\text{snc}}} e_2, e_1)$

Proof.
Similar to the proof of law 3.3.16. □

Law 3.3.19 If e_1 does not depend on y and e_2 does not depend on x then:

$$(x \overset{:=}{_{\text{snc}}} e_1; y \overset{:=}{_{\text{snc}}} e_2) = (y \overset{:=}{_{\text{snc}}} e_2; x \overset{:=}{_{\text{snc}}} e_1)$$

Proof.
Similar to the proof of law 3.3.16. □

Theorem 3.3.20 Selection of **S** designs

$$\mathbf{S}(P) \lhd b \rhd \mathbf{S}(Q) = \mathbf{S}(P \lhd b \rhd Q)$$

Proof.

$$
\begin{aligned}
& \mathbf{S}(P) \lhd b \rhd \mathbf{S}(Q) \\
= & \quad \{\text{Definition 2.2.1}\}
\end{aligned}
$$

$$
\begin{array}{ll}
 & (b \wedge \mathbf{S}(P)) \vee (\neg b \wedge \mathbf{S}(Q)) \\
= & \{\text{Theorem 3.2.14}\} \\
 & \mathbf{S}(b \wedge P) \vee \mathbf{S}(\neg b \wedge Q) \\
= & \{\text{Theorem 3.2.4}\} \\
 & \mathbf{S}((b \wedge P) \vee (\neg b \wedge Q)) \\
= & \{\text{Definition 2.2.1}\} \\
 & \mathbf{S}(P \lhd b \rhd Q)
\end{array}
$$

□

Theorem 3.3.21 Design characterisation of **S** selection

$$\mathbf{S}(P_1 \vdash P_2) \lhd b \rhd \mathbf{S}(Q_1 \vdash Q_2) = \mathbf{S}(P_1 \lhd b \rhd Q_1 \vdash P_2 \lhd b \rhd Q_2)$$

Proof.

$$
\begin{array}{ll}
 & \mathbf{S}(P_1 \vdash P_2) \lhd b \rhd \mathbf{S}(Q_1 \vdash Q_2) \\
= & \{\text{Theorem 3.3.20}\} \\
 & \mathbf{S}((P_1 \vdash P_2) \lhd b \rhd (Q_1 \vdash Q_2)) \\
= & \{\text{Theorem 2.3.18}\} \\
 & \mathbf{S}(P_1 \lhd b \rhd Q_1 \vdash P_2 \lhd b \rhd Q_2)
\end{array}
$$

□

Law 3.3.22 $P \lhd b \rhd Q = Q \lhd \neg b \rhd P$

Proof.

$$
\begin{array}{ll}
 & \mathbf{S}(P) \lhd b \rhd \mathbf{S}(Q) \\
= & \{\text{Theorem 3.3.20}\} \\
 & \mathbf{S}(P \lhd b \rhd Q) \\
= & \{\text{Law 2.3.23}\} \\
 & \mathbf{S}(Q \lhd \neg b \rhd P) \\
= & \{\text{Theorem 3.3.20}\} \\
 & \mathbf{S}(Q) \lhd \neg b \rhd \mathbf{S}(P)
\end{array}
$$

□

Law 3.3.23 $P \lhd b \rhd P = P$

Law 3.3.24 $P \lhd \mathbf{true} \rhd Q = P$

Law 3.3.25 $P \lhd \mathbf{false} \rhd Q = Q$

Law 3.3.26 $(P \lhd b \rhd Q) \lhd b \rhd R = P \lhd b \rhd R = P \lhd b \rhd (Q \lhd b \rhd R)$

Law 3.3.27 $(P \lhd b \rhd Q) \lhd c \rhd R = (P \lhd b \wedge c \rhd (Q \lhd c \rhd R))$

Law 3.3.28 $x \mathrel{:=}_{\text{snc}} e_1 \lhd b \rhd x \mathrel{:=}_{\text{snc}} e_2 = x \mathrel{:=}_{\text{snc}} (e_1 \lhd b \rhd e_2)$

Law 3.3.29 $(P \lhd c \rhd Q) \lhd b \rhd R = (P \lhd b \rhd R) \lhd c \rhd (Q \lhd b \rhd R)$

Law 3.3.30 $P \lhd b \rhd (P \lhd c \rhd Q) = P \lhd (b \vee c) \rhd Q$

Law 3.3.31 $\neg(s \wedge b) \wedge (P \lhd (s \wedge \neg b) \rhd Q) = \neg(s \wedge b) \wedge (P \lhd s \rhd Q)$

Proof.

The proofs of laws 3.3.23 to 3.3.31 follow the same approach used to prove law 3.3.22 (i.e., we extract **S** by means of theorem 3.3.20 and then apply the equivalent design-equivalent law to complete the proof). □

Law 3.3.32 $x \mathrel{:=}_{\text{snc}} e; (P \lhd b(x) \rhd Q) = (x \mathrel{:=}_{\text{snc}} e; P) \lhd b(e) \rhd (x \mathrel{:=}_{\text{snc}} e; Q)$

Proof.

$$
\begin{aligned}
& x \mathrel{:=}_{\text{snc}} e; (P \lhd b(x) \rhd Q) \\
= \; & \{\text{Theorem 3.3.21 and definition 2.3.9}\} \\
& \mathbf{S}(\mathbf{true} \vdash x := e); \mathbf{S}(P_1 \lhd b(x) \rhd Q_1 \vdash P_2 \lhd b(x) \rhd Q_2) \\
= \; & \{\text{Theorem 3.3.6 and propositional calculus}\} \\
& \mathbf{S}(\neg(\mathbf{S}(x := e); \mathbf{S}(\neg(P_1 \lhd b(x) \rhd Q_1))) \vdash \mathbf{S}(x := e); \mathbf{S}(P_2 \lhd b(x) \rhd Q_2)) \\
= \; & \{\text{Propositional calculus, theorems 3.2.4 and 3.3.14}\} \\
& \mathbf{S}(\neg(x := e; (\mathbf{S}(\neg P_1) \lhd b(x) \rhd \mathbf{S}(\neg Q_1))) \vdash x := e; (\mathbf{S}(P_2) \lhd b \rhd \mathbf{S}(Q_2))) \\
= \; & \{\text{Law 2.3.25}\} \\
& \mathbf{S}(\neg((x := e; \mathbf{S}(\neg P_1)) \lhd b(e) \rhd (x := e; \mathbf{S}(Q_1))) \vdash (x := e; \mathbf{S}(P_2)) \lhd b(e) \rhd (x := e; \mathbf{S}(Q_2))) \\
= \; & \{\text{Propositional calculus}\} \\
& \mathbf{S}(\neg(x := e; \mathbf{S}(\neg P_1)) \lhd b(e) \rhd \neg(x := e; \mathbf{S}(\neg Q_1)) \vdash (x := e; \mathbf{S}(P_2)) \lhd b(e) \rhd (x := e; \mathbf{S}(Q_2))) \\
= \; & \{\text{Theorems 3.3.14 and 3.3.21}\} \\
& \mathbf{S}(\neg(\mathbf{S}(x := e); \mathbf{S}(\neg P_1)) \vdash \mathbf{S}(x := e); \mathbf{S}(P_2)) \lhd b(e) \rhd \\
& \quad \mathbf{S}(\neg(\mathbf{S}(x := e); \mathbf{S}(\neg Q_1)) \vdash \mathbf{S}(x := e); \mathbf{S}(Q_2)) \\
= \; & \{\text{Definition 2.3.9, theorem 3.3.6 and propositional calculus}\} \\
& (x \mathrel{:=}_{\text{snc}} e; P) \lhd b(e) \rhd (x \mathrel{:=}_{\text{snc}} e; Q)
\end{aligned}
$$

□

Law 3.3.33 $(P \lhd b \rhd Q); R = (P; R) \lhd b \rhd (Q; R)$

Proof.

$$
\begin{aligned}
& (P \lhd b \rhd Q); R \\
= \; & \{\text{Definition 2.2.1}\}
\end{aligned}
$$

$(b \wedge P) \vee (\neg b \wedge Q); R$

= {Disjunctivity of sequential composition}

$(b \wedge P; R) \vee (\neg b \wedge Q; R)$

= {Definition 2.2.3}

$(\exists v_0 \bullet b[v_0/v'] \wedge P[v_0/v'] \wedge R[v_0/v]) \vee (\exists v_0 \bullet \neg b[v_0/v'] \wedge Q[v_0/v'] \wedge R[v_0/v])$

= {Condition does not refer to dashed variables, quantifier contract scope}

$(b \wedge \exists v_0 \bullet P[v_0/v'] \wedge R[v_0/v]) \vee (\neg b \wedge \exists v_0 \bullet Q[v_0/v'] \wedge R[v_0/v])$

= {Definition 2.2.3}

$(b \wedge (P; R)) \vee (\neg b \wedge (Q; R))$

= {Definition 2.2.1}

$(P; R) \lhd b \rhd (Q; R)$ □

Theorem 3.3.36 Design assumption – synchronous assumption equivalence

$$b_{\mathsf{S}}^{\top} = \mathbf{S}(b^{\top})$$

Proof.

$\mathbf{S}(b^{\top})$

= {Theorems 2.3.52 and 3.3.20}

$\mathbf{S}(I\!I_D) \lhd b \rhd \mathbf{S}(\neg ok)$

= {Definitions 3.2.7 and 3.3.1}

$\text{I\!I} \lhd b \rhd \top$

= {Definition 3.3.34}

$b_{\mathsf{S}}^{\top}$ □

Theorem 3.3.37 Design assertion – synchronous assertion equivalence

$$b_{\mathsf{S}}^{\top} = \mathbf{S}(b^{\top})$$

Proof.

$\mathbf{S}(b_{\perp})$

= {Theorems 2.3.53 and 3.3.20}

$\mathbf{S}(I\!I_D) \lhd b \rhd \mathbf{S}(\mathbf{true})$

= {Definitions 3.2.7 and 3.3.2}

$\text{I\!I} \lhd b \rhd \perp$

= {Definition 3.3.34}

$b_{\mathrm{S}}^{\bot}$ □

Law 3.3.38 $(\mathbf{false})_{\mathrm{S}}^{\top} = \top$

Proof.

$(\mathbf{false})_{\mathrm{S}}^{\top}$
= {Theorem 3.3.36}
$\mathbf{S}(\mathbf{false}^{\top})$
= {Law 2.3.54}
$\mathbf{S}(\neg ok)$
= {Definition 3.3.1}
$\top$ □

Law 3.3.39 $(\mathbf{false})_{\mathrm{S}}^{\bot} = \bot$

Proof.
Similar to proof of law 3.3.38. □

Law 3.3.40 $b_{\mathrm{S}}^{\top} \sqsupseteq \text{Ⅱ} \sqsupseteq b_{\mathrm{S}}^{\bot}$

Proof.

$b_{\mathrm{S}}^{\top}$
= {Definition 3.3.34}
$\text{Ⅱ} \lhd b \rhd \top$
$\sqsupseteq$ {Law 3.3.4}
$\text{Ⅱ} \lhd b \rhd \text{Ⅱ}$
= {Law 3.3.23}
Ⅱ
$\sqsupseteq$ {Laws 3.3.23 and 3.3.5}
$\text{Ⅱ} \lhd b \rhd \bot$
= {Definition 3.3.35}
$b_{\mathrm{S}}^{\bot}$ □

Law 3.3.41 $b_{\mathrm{S}}^{\top}; b_{\mathrm{S}}^{\bot} = b_{\mathrm{S}}^{\top}$

Proof.

$b_{\mathrm{S}}^{\top}; b_{\mathrm{S}}^{\bot}$
= {Definition 3.3.34, laws 3.3.33, 3.3.9 and 3.3.7}

$b^{\perp}_{\mathsf{S}} \lhd b \rhd \top$

= {Definition 3.3.35 and law 3.3.26}

$\mathbb{II} \lhd b \rhd \top$

= {Definition 3.3.34}

$b^{\top}_{\mathsf{S}}$ □

Law 3.3.42 $b^{\perp}_{\mathsf{S}}; b^{\top}_{\mathsf{S}} = b^{\perp}_{\mathsf{S}}$

Proof.

Similar to the proof of law 3.3.41. □

Law 3.3.43 $b^{\top}_{\mathsf{S}}; b^{\perp}_{\mathsf{S}} \sqsupseteq \mathbb{II}$

Proof. Straightforward from laws 3.3.41 and 3.3.40. □

Law 3.3.44 $b^{\perp}_{\mathsf{S}}; b^{\top}_{\mathsf{S}} \sqsubseteq \mathbb{II}$

Proof. Straightforward from laws 3.3.42 and 3.3.40. □

Law 3.3.45 $b^{\top}_{\mathsf{S}}; c^{\top}_{\mathsf{S}} = (b \wedge c)^{\top}_{\mathsf{S}}$

Proof.

$b^{\top}_{\mathsf{S}}; c^{\top}_{\mathsf{S}}$

= {Definition 3.3.34}

$(\mathbb{II} \lhd b \rhd \top); (\mathbb{II} \lhd c \rhd \top)$

= {Laws 3.3.33, 3.3.9 and 3.3.7}

$(\mathbb{II} \lhd c \rhd \top) \lhd b \rhd \top$

= {Law 3.3.27}

$\mathbb{II} \lhd c \wedge b \rhd (\top \lhd b \rhd \top)$

= {Law 3.3.23, definition 3.3.34 and propositional calculus}

$(b \wedge c)^{\top}_{\mathsf{S}}$ □

Law 3.3.46 $b^{\top}_{\mathsf{S}}; c^{\top}_{\mathsf{S}} = c^{\top}_{\mathsf{S}}; b^{\top}_{\mathsf{S}}$

Proof.

Direct from law 3.3.45 and commutativity of conjunction. □

Law 3.3.47 $b^{\perp}_{\mathsf{S}}; c^{\perp}_{\mathsf{S}} = (b \wedge c)^{\perp}_{\mathsf{S}}$

Proof.

Similar to the proof of law 3.3.45. □

Law 3.3.48 $(b \vee c)^{\top}_{\mathsf{S}}; b^{\top}_{\mathsf{S}} = b^{\top}_{\mathsf{S}}$

Proof.

$$
\begin{aligned}
& (b \vee c)^{\top}_{\mathrm{S}} ; b^{\top}_{\mathrm{S}} \\
= & \quad \{\text{Law 3.3.45}\} \\
& ((b \vee c) \wedge b)^{\top}_{\mathrm{S}} \\
= & \quad \{\text{Propositional calculus}\} \\
& b^{\top}_{\mathrm{S}}
\end{aligned}
$$

□

Law 3.3.49 $b^{\bot}_{\mathrm{S}} ; b^{\bot}_{\mathrm{S}} = b^{\bot}_{\mathrm{S}}$

Proof.
Similar to the proof of law 3.3.48. □

Law 3.3.50 $b^{\top}_{\mathrm{S}} ; b^{\top}_{\mathrm{S}} = b^{\top}_{\mathrm{S}}$

Proof.
Similar to the proof of law 3.3.48. □

Law 3.3.51 $b^{\top}_{\mathrm{S}} ; \mathrm{II} = b^{\top}_{\mathrm{S}}$

Proof.

$$
\begin{aligned}
& b^{\top}_{\mathrm{S}} ; \mathrm{II} \\
= & \quad \{\text{Definition 3.3.34 and law 3.3.33}\} \\
& (\mathrm{II} ; \mathrm{II}) \lhd b \rhd (\top ; \mathrm{II}) \\
= & \quad \{\text{Laws 3.3.9 and 3.3.7, definition 3.3.34}\} \\
& b^{\top}_{\mathrm{S}}
\end{aligned}
$$

□

Law 3.3.52 $b^{\bot}_{\mathrm{S}} ; \mathrm{II} = b^{\bot}_{\mathrm{S}}$

Proof.
Similar to the proof of law 3.3.51. □

Law 3.3.53 Provided e_1 does not mention x or y we have:

$$(x, y \overset{:=}{_{\mathrm{snc}}} e_1, e_2) = (x, y \overset{:=}{_{\mathrm{snc}}} e_1, e_2) ; (x = e_1)^{\top}_{\mathrm{S}}$$

Proof.

$$
\begin{aligned}
& (x, y \overset{:=}{_{\mathrm{snc}}} e_1, e_2) ; (x = e_1)^{\top}_{\mathrm{S}} \\
= & \quad \{\text{Definition 3.3.34}\} \\
& (x, y \overset{:=}{_{\mathrm{snc}}} e_1, e_2) ; (\mathrm{II} \lhd x = e_1 \rhd \top) \\
= & \quad \{\text{Law 3.3.32}\}
\end{aligned}
$$

$$
\begin{array}{ll}
 & (x, y \mathrel{:=}_{\mathrm{snc}} e_1, e_2; \mathrm{II}) \lhd (x = e_1)[e_1, e_2/x, y] \rhd (x, y \mathrel{:=}_{\mathrm{snc}} e_1, e_2; \top) \\
= & \{\text{Assumption } (e_1 \text{ does not mention } x \text{ or } y), \text{propositional calculus}\} \\
 & (x, y \mathrel{:=}_{\mathrm{snc}} e_1, e_2; \mathrm{II}) \lhd e_1 = e_1 \rhd (x, y \mathrel{:=}_{\mathrm{snc}} e_1, e_2; \top) \\
= & \{\text{Propositional calculus, law 3.3.24}\} \\
 & x, y \mathrel{:=}_{\mathrm{snc}} e_1, e_2; \mathrm{II} \\
= & \{\text{Theorem 3.3.15, law 3.3.10}\} \\
 & x, y \mathrel{:=}_{\mathrm{snc}} e_1, e_2
\end{array}
$$

□

Law 3.3.54 Provided e_1 does not mention x or y we have:

$$(x, y \mathrel{:=}_{\mathrm{snc}} e_1, e_2) = (x, y \mathrel{:=}_{\mathrm{snc}} e_1, e_2); (x = e_1)^{\perp}_{\mathrm{S}}$$

Proof.

Similar to the proof of law 3.3.53. □

Law 3.3.55 $(x \mathrel{:=}_{\mathrm{snc}} y; (y = e)^{\perp}_{\mathrm{S}}) = (x \mathrel{:=}_{\mathrm{snc}} y; (x = e)^{\perp}_{\mathrm{S}})$

Proof.

$$
\begin{array}{ll}
 & x \mathrel{:=}_{\mathrm{snc}} y; (y = e)^{\perp}_{\mathrm{S}} \\
= & \{\text{Theorems 3.3.37 and 2.3.53, definition 3.3.13}\} \\
 & \mathbf{S}(\mathbf{true} \vdash x := y); \mathbf{S}(y = e \vdash I\!I) \\
= & \{\text{Theorem 3.3.6 and propositional calculus}\} \\
 & \mathbf{S}(\neg(\mathbf{S}(x := y); \mathbf{S}(\neg y = e)) \vdash \mathbf{S}(x := y); \mathbf{S}(I\!I)) \\
= & \{\text{Theorem 3.3.14, definitions 2.2.3 and 3.2.2, propositional calculus}\} \\
 & \mathbf{S}(\neg(\neg(y = e[y/x]) \wedge \mathbf{S}[y/x]) \vdash \mathbf{S}(x := y); \mathbf{S}(I\!I)) \\
= & \{\text{Propositional calculus}\} \\
 & \mathbf{S}(\neg(\neg(x = e[y/x]) \wedge \mathbf{S}[y/x]) \vdash \mathbf{S}(x := y); \mathbf{S}(I\!I)) \\
= & \{\text{Inverse steps}\} \\
 & \mathbf{S}(\neg(\mathbf{S}(x := y); \mathbf{S}(\neg x = e)) \vdash \mathbf{S}(x := y); \mathbf{S}(I\!I)) \\
= & \{\text{Theorem 3.3.6 and propositional calculus}\} \\
 & \mathbf{S}(\mathbf{true} \vdash x := y); \mathbf{S}(x = e \vdash I\!I) \\
= & \{\text{Theorems 3.3.37 and 2.3.53, definition 3.3.13}\} \\
 & x \mathrel{:=}_{\mathrm{snc}} y; (x = e)^{\perp}_{\mathrm{S}}
\end{array}
$$

□

Law 3.3.56 Provided b does not depend on x we have:

$$x \mathrel{:=}_{\mathrm{snc}} e; b^{\top}_{\mathrm{S}} = b^{\top}_{\mathrm{S}}; x \mathrel{:=}_{\mathrm{snc}} e$$

Proof.

$$
\begin{aligned}
& x \mathrel{:=}_{\mathrm{snc}} e; b_{\mathrm{S}}^{\top} \\
= & \{\text{Definition 3.3.34, law 3.3.32}\} \\
& (x \mathrel{:=}_{\mathrm{snc}} e; \text{Ⅱ}) \lhd b(x) \rhd (x \mathrel{:=}_{\mathrm{snc}} e; \top) \\
= & \{\text{Assumption } (b \text{ does not mention } x), \text{ law 3.3.10 and predicate calculus}\} \\
& x \mathrel{:=}_{\mathrm{snc}} e \lhd b \rhd \top \\
= & \{\text{Laws 3.3.9, 3.3.7 and 3.3.33}\} \\
& (\text{Ⅱ} \lhd b \rhd \top); x \mathrel{:=}_{\mathrm{snc}} e \\
= & \{\text{Definition 3.3.34}\} \\
& b_{\mathrm{S}}^{\top}; x \mathrel{:=}_{\mathrm{snc}} e
\end{aligned}
$$

□

Law 3.3.57 $P \lhd b \rhd Q = (b_{\mathrm{S}}^{\top}; P) \lhd b \rhd ((\neg b)_{\mathrm{S}}^{\top}; Q)$

Proof.

$$
\begin{aligned}
& (b_{\mathrm{S}}^{\top}; P) \lhd b \rhd ((\neg b)_{\mathrm{S}}^{\top}; Q) \\
= & \{\text{Definition 3.3.34}\} \\
& ((\text{Ⅱ} \lhd b \rhd \top); P) \lhd b \rhd ((\text{Ⅱ} \lhd \neg b \rhd \top); Q) \\
= & \{\text{Laws 3.3.33, 3.3.9 and 3.3.7}\} \\
& (P \lhd b \rhd \top) \lhd b \rhd (Q \lhd \neg b \rhd \top) \\
= & \{\text{Law 3.3.26}\} \\
& P \lhd b \rhd Q
\end{aligned}
$$

□

Law 3.3.58 $b_{\mathrm{S}}^{\top}; (P \lhd b \rhd Q) = b_{\mathrm{S}}^{\top}; P$

Proof.

$$
\begin{aligned}
& b_{\mathrm{S}}^{\top}; (P \lhd b \rhd Q) \\
= & \{\text{Definition 3.3.34, laws 3.3.33, 3.3.9 and 3.3.7}\} \\
& (P \lhd b \rhd Q) \lhd b \rhd \top \\
= & \{\text{Laws 3.3.26, 3.3.9 and 3.3.7}\} \\
& (\text{Ⅱ}; P) \lhd b \rhd (\top; P) \\
= & \{\text{Law 3.3.33, definition 3.3.34}\} \\
& b_{\mathrm{S}}^{\top}; P
\end{aligned}
$$

□

Law 3.3.59 $(\neg b)_{\mathrm{S}}^{\top}; (P \lhd b \rhd Q) = (\neg b)_{\mathrm{S}}^{\top}; Q$

Proof.

Similar to the proof of law 3.3.58. □

Law 3.3.60 Provided $(b \wedge c = \mathbf{false})$ we have:

$$b_{\mathrm{S}}^{\top}; (P \lhd c \rhd Q) = (b_{\mathrm{S}}^{\top}; Q)$$

Proof.

$$\begin{array}{ll} & b_{\mathrm{S}}^{\top}; (P \lhd c \rhd Q) \\ = & \{\text{Definition 3.3.34, laws 3.3.33, 3.3.9 and 3.3.7}\} \\ & (P \lhd c \rhd Q) \lhd b \rhd \top \\ = & \{\text{Law 3.3.27}\} \\ & (P \lhd c \wedge b \rhd Q) \lhd b \rhd \top \\ = & \{\text{Assumption, law 3.3.25}\} \\ & Q \lhd b \rhd \top \\ = & \{\text{Laws 3.3.9, 3.3.7 and 3.3.33, then definition 3.3.34}\} \\ & b_{\mathrm{S}}^{\top}; Q \end{array}$$

□

Law 3.3.61 Provided P and Q are **S**-healthy we have:

$$c_{\mathrm{S}}^{\top}; (P \lhd b \wedge c \rhd Q) = c_{\mathrm{S}}^{\top}; (P \lhd b \rhd Q)$$

Proof.

$$\begin{array}{ll} & c_{\mathrm{S}}^{\top}; (P \lhd b \rhd Q) \\ = & \{\text{Definition 3.3.34, laws 3.3.33, 3.3.9 and 3.3.7}\} \\ & (P \lhd b \rhd Q) \lhd c \rhd \top \\ = & \{\text{Law 3.3.27}\} \\ & (P \lhd b \wedge c \rhd Q) \lhd c \rhd \top \\ = & \{\text{Laws 3.3.9, 3.3.7 and 3.3.33, definition 3.3.34}\} \\ & c_{\mathrm{S}}^{\top}; (P \lhd b \wedge c \rhd Q) \end{array}$$

□

Law 3.3.62 $b_{\mathrm{S}}^{\top}; (P \lhd c \rhd Q) = (b_{\mathrm{S}}^{\top}; P) \lhd c \rhd (b_{\mathrm{S}}^{\top}; Q)$

Proof.

$$\begin{array}{ll} & b_{\mathrm{S}}^{\top}; (P \lhd c \rhd Q) \\ = & \{\text{Definition 3.3.34, laws 3.3.33, 3.3.9 and 3.3.7}\} \\ & (P \lhd c \rhd Q) \lhd b \rhd \top \\ = & \{\text{Definition 2.2.1, propositional calculus}\} \\ & ((c \wedge b \wedge P) \vee (\neg c \wedge b \wedge Q) \vee (\neg b \wedge \top) \\ = & \{\text{Propositional calculus}\} \end{array}$$

$$
\begin{array}{ll}
 & ((c \wedge b \wedge ((b \wedge P) \vee (\neg b \wedge \top))) \vee (\neg c \wedge b \wedge ((b \wedge Q) \vee (\neg b \wedge \top))) \vee (\neg b \wedge \top) \\
= & \{\text{Definition 2.2.1, propositional calculus}\} \\
 & (b \wedge (c \wedge (P \lhd b \rhd \top)) \vee (\neg c \wedge (Q \lhd b \rhd \top))) \vee (\neg b \wedge \top) \\
= & \{\text{Definition 2.2.1}\} \\
 & ((P \lhd b \rhd \top) \lhd c \rhd (Q \lhd b \rhd \top)) \lhd b \rhd \top \\
= & \{\text{Laws 3.3.9, 3.3.7 and 3.3.33, then definition 3.3.34}\} \\
 & (b_{\mathsf{S}}^{\top}; P) \lhd c \rhd (b_{\mathsf{S}}^{\top}; Q)
\end{array}
$$

□

Law 3.3.63 $b_{\mathsf{S}}^{\bot}; (P \lhd c \rhd Q) = (b_{\mathsf{S}}^{\bot}; P) \lhd c \rhd (b_{\mathsf{S}}^{\bot}; Q)$

Proof.

Similar to the proof of law 3.3.62. □

Theorem 3.3.66 Provided P is **S**-healthy and x is not an observational variable we have:

$$\mathtt{var}\ x; P = \exists x \bullet P$$

Proof.

$$
\begin{array}{ll}
 & \mathtt{var}\ x; P \\
= & \{\text{Definition 3.3.64, theorem 3.3.6 and propositional calculus}\} \\
 & \mathbf{S}(\neg(\mathbf{S}(\mathbf{var}\ x); \mathbf{S}(\neg P_1)) \vdash \mathbf{S}(\mathbf{var}\ x); \mathbf{S}(P_2)) \\
= & \{\text{Theorem 3.3.68, definitions 2.2.18 and 2.3.1}\} \\
 & \mathbf{S}(\neg ok \vee \exists x \bullet \mathbf{S}(\neg P_1) \vee (ok' \wedge \exists x \bullet \mathbf{S}(P_2))) \\
= & \{\text{Assumption: } x \text{ is not an observational variable, quantifier expand scope, exists-associativity}\} \\
 & \exists x \bullet \mathbf{S}(\neg ok \vee \mathbf{S}(\neg P_1) \vee (ok' \wedge \mathbf{S}(P_2))) \\
= & \{\text{Theorems 3.2.14, 3.2.13 and } \mathbf{S} \text{ itempotent}\} \\
 & \exists x \bullet \mathbf{S}(\neg ok \vee \neg P_1 \vee (ok' \wedge P_2)) \\
= & \{\text{Propositional calculus, definition 2.3.1 and assumption } (P \text{ is } \mathbf{S})\} \\
 & \exists x \bullet P
\end{array}
$$

□

Theorem 3.3.67 Provided P is **SH3** $\circ$ **S**-healthy and P_1 does not mention x', and x is not an observational variable we have:

$$P; \mathsf{end}\ x = \exists x' \bullet P$$

Proof.

$$
\begin{array}{ll}
 & P; \mathsf{end}\ x \\
= & \{\text{Definition 3.3.65, theorem 3.3.6 and propositional calculus}\}
\end{array}
$$

$\mathbf{S}(\neg\mathbf{S}(\neg P_1) \vdash \mathbf{S}(P_2); \mathbf{S}(\mathbf{end}\ x))$

= {Theorem 3.3.69, definitions 2.2.19 and 2.3.1}

$\mathbf{S}(\neg ok \vee \mathbf{S}(\neg P_1) \vee (ok' \wedge \exists x' \bullet \mathbf{S}(P_2)))$

= {Assumption, propositional calculus}

$\exists x' \bullet \mathbf{S}(\neg ok \vee \mathbf{S}(\neg P_1) \vee (ok' \wedge \mathbf{S}(P_2)))$

= {Theorems 3.2.14, 3.2.13 and **S** itempotent}

$\exists x' \bullet \mathbf{S}(\neg ok \vee \neg P_1 \vee (ok' \wedge P_2))$

= {Propositional calculus, definition 2.3.1 and assumption (P is **S**)}

$\exists x' \bullet P$ □

Theorem 3.3.68 $\mathbf{S}(\mathbf{var}\ x) = \mathbf{var}\ x$

Proof.

The variables introduced with the dynamic scope operators cannot be observational variables (*ok*, *x.out*, *x.in*, *c* and their dashed variables in our theory), hence the proof is similar to the proof of theorem 3.2.9. □

Theorem 3.3.69 $\mathbf{S}(\mathbf{end}\ x) = \mathbf{end}\ x$

Proof.

The variables introduced with the dynamic scope operators cannot be observational variables (*ok*, *x.out*, *x.in*, *c* and their dashed variables in our theory), hence the proof is similar to the proof of theorem 3.2.9. □

Law 3.3.70 $\mathtt{var}\ x; \mathtt{end}\ x = \text{Ⅱ}$

Proof.

$\mathtt{var}\ x; \mathtt{end}\ x$

= {Definitions 3.3.64 and 3.3.65}

$\mathbf{S}(\mathbf{var}\ x); \mathbf{S}(\mathbf{end}\ x)$

= {Definitions 2.3.35 and 2.3.36}

$\mathbf{S}(\mathbf{true} \vdash \mathbf{var}\ x); \mathbf{S}(\mathbf{true} \vdash \mathbf{end}\ x)$

= {Theorem 3.3.6 and propositional calculus}

$\mathbf{S}(\mathbf{true} \vdash \mathbf{S}(\mathbf{var}\ x); \mathbf{S}(\mathbf{end}\ x))$

= {Theorems 3.3.68 and 3.3.69, law 2.3.37}

$\mathbf{S}(\mathbf{true} \vdash I\!I)$

= {Definitions 2.3.8 and 3.2.7}

Ⅱ □

Law 3.3.71 $\mathtt{end}\ x; \mathtt{var}\ x \sqsubseteq \text{Ⅱ}$

Proof.

Similar to the proof of law 3.3.70, but using law 2.3.38 instead of 2.3.37. □

Law 3.3.72 $x \mathrel{:=}_{\mathrm{snc}} e; \mathtt{end}\ x = \mathtt{end}\ x$

Proof.

$$
\begin{aligned}
& x \mathrel{:=}_{\mathrm{snc}} e; \mathtt{end}\ x \\
= & \ \{\text{Definitions 3.3.13, 3.3.65 and 2.3.36}\} \\
& \mathbf{S}(\mathbf{true} \vdash x := e); \mathbf{S}(\mathbf{true} \vdash \mathbf{end}\ x) \\
= & \ \{\text{Theorem 3.3.6 and propositional calculus}\} \\
& \mathbf{S}(\mathbf{true} \vdash \mathbf{S}(x := e); \mathbf{S}(\mathbf{end}\ x)) \\
= & \ \{\text{Theorems 3.3.14 and 3.3.69, law 2.3.40}\} \\
& \mathbf{S}(\mathbf{true} \vdash \mathbf{end}\ x) \\
= & \ \{\text{Definition 3.3.65}\} \\
& \mathtt{end}\ x
\end{aligned}
$$

□

Law 3.3.73 $\mathtt{var}\ x \sqsubseteq (\mathtt{var}\ x; x \mathrel{:=}_{\mathrm{snc}} e)$

Proof.

Similar to the proof of law 3.3.70, but using law 2.3.41 instead of 2.3.37. □

Law 3.3.74 $\mathtt{var}\ x; (x = e)^{\top}_{\mathrm{S}} = (\mathtt{var}\ x; x \mathrel{:=}_{\mathrm{snc}} e)$

Proof.

$$
\begin{aligned}
& \mathtt{var}\ x; (x = e)^{\top}_{\mathrm{S}} \\
= & \ \{\text{Definitions 3.3.64 and theorem 3.3.36}\} \\
& \mathbf{S}(\mathbf{true} \vdash \mathbf{var}\ x); \mathbf{S}(\mathbf{true} \vdash I\!I \lhd x = e \rhd \mathbf{false}) \\
= & \ \{\text{Theorem 3.3.6 and propositional calculus}\} \\
& \mathbf{S}(\mathbf{true} \vdash \mathbf{S}(\mathbf{var}\ x); \mathbf{S}(I\!I \lhd x = e \rhd \mathbf{false})) \\
= & \ \{\text{Theorems 3.3.20, 3.3.68, 3.3.14 and propositional calculus}\} \\
& \mathbf{S}(\mathbf{true} \vdash \mathbf{var}\ x; (I\!I \lhd x = e \rhd \mathbf{false})) \\
= & \ \{\text{Definition 2.3.50 and law 2.3.79, propositional calculus}\} \\
& \mathbf{S}(\mathbf{true} \vdash \mathbf{var}\ x; x := e) \\
= & \ \{\text{Theorems 3.3.68 and 3.3.14, propositional calculus}\} \\
& \mathbf{S}(\neg(\mathbf{S}(\mathbf{var}\ x); \mathbf{S}(\mathbf{false})) \vdash \mathbf{S}(\mathbf{var}\ x); \mathbf{S}(x := e)) \\
= & \ \{\text{Theorem 3.3.6 and 3.3.14, propositional calculus and definitions 3.3.13 3.3.64}\} \\
& \mathtt{var}\ x; x \mathrel{:=}_{\mathrm{snc}} e
\end{aligned}
$$

□

Law 3.3.75 $\texttt{var}\ x; \texttt{var}\ y = \texttt{var}\ x, y$

Proof.

$\texttt{var}\ x; \texttt{var}\ y$
= {Definition 3.3.64, theorem 3.3.6 and propositional calculus}
$\mathbf{S}(\mathbf{true} \vdash \mathbf{S}(\mathbf{var}\ x); \mathbf{S}(\mathbf{var}\ y))$
= {Theorem 3.3.68 and law 2.3.42}
$\mathbf{S}(\mathbf{true} \vdash \mathbf{var}\ x, y)$
= {Definition 3.3.64}
$\texttt{var}\ x, y$ □

Law 3.3.76 $\texttt{end}\ x; \texttt{end}\ y = \texttt{end}\ x, y$

Proof.
Similar to proof of law 3.3.75. □

Law 3.3.77 $\texttt{var}\ x, y = \texttt{var}\ y, x$

Proof.
Similar to proof of law 3.3.75. □

Law 3.3.78 $\texttt{end}\ x, y = \texttt{end}\ y, x$

Proof. Similar to proof of law 3.3.75. □

Law 3.3.79 If x is not free in e then

$$(\texttt{end}\ x; \texttt{var}\ x \mathrel{:=}_{\text{snc}} e) = (x \mathrel{:=}_{\text{snc}} e)$$

Proof.

$\texttt{end}\ x; \texttt{var}\ x \mathrel{:=}_{\text{snc}} e$
= {Definitions 3.3.65, 3.3.64 and theorem 3.3.6}
$\mathbf{S}(\mathbf{true} \vdash \mathbf{S}(\mathbf{end}\ x); \mathbf{S}(\mathbf{var}\ x); \mathbf{S}(x := e))$
= {Theorems 3.3.14, 3.3.68 and 3.3.69}
$\mathbf{S}(\mathbf{true} \vdash \mathbf{end}\ x; \mathbf{var}\ x; x := e)$
= {Law 2.3.39 and definition 3.3.13}
$x \mathrel{:=}_{\text{snc}} e$ □

Law 3.3.80 Provided P is **SH3** and neither e nor $\mathbf{S}$ mention x we have:

$$(P; x \mathrel{:=}_{\text{snc}} e) = ((P; \texttt{end}\ x)_{+x}; x \mathrel{:=}_{\text{snc}} e)$$

Proof.

Similar to the proof of law 2.3.49. □

Law 3.3.81 Provided b does not mention x we have:

$$\texttt{var}\ x; (P \lhd b \rhd Q) = (\texttt{var}\ x; P) \lhd b \rhd (\texttt{var}\ x; Q)$$

Proof.

$$\begin{array}{ll}
& \texttt{var}\ x; (P \lhd b \rhd Q) \\
= & \{\text{Definition 2.2.1 and theorem 3.3.66}\} \\
& \exists x \bullet (b \wedge P) \vee (\neg b \wedge Q) \\
= & \{\text{Quantifier associativity}\} \\
& (\exists x \bullet b \wedge P) \vee (\exists x \bullet \neg b \wedge Q) \\
= & \{\text{Assumption: } b \text{ does not mention } x\text{, quantifier contract scope}\} \\
& (b \wedge \exists x \bullet P) \vee (\neg b \wedge \exists x \bullet Q) \\
= & \{\text{Definition 2.2.1 and theorem 3.3.66}\} \\
& (\texttt{var}\ x; P) \lhd b \rhd (\texttt{var}\ x; Q)
\end{array}$$

□

Law 3.3.85 Provided $P = (P_1 \vdash P_2)$ is a **SH3**, **S**-healthy design we have:

$$\mathbf{S}(P)_{[x]} = \mathbf{S}(P_1 \vdash P2; \mathbf{var}\ 0.c := c; \mathbf{end}\ c)_{+x,c}; E_D(x)$$

where $E_S(x)$ is defined as follows:

$$E_S(x) =_{df} \mathbf{S}(\mathbf{true} \vdash (E(x, 0.c) \wedge c := 0.c); \mathbf{end}\ 0.c)$$

Proof.

$$\begin{array}{ll}
& \mathbf{S}(P_1 \vdash P2; \mathbf{var}\ 0.c := c; \mathbf{end}\ c)_{+x,c}; E_D(x) \\
= & \{\text{Definitions 2.2.3, 2.2.18, 2.2.19 and predicate calculus}\} \\
& \mathbf{S}(P_1 \vdash P2[0.c'/c'] \wedge c', x' = c, x); E_D(x) \\
= & \{\text{Definition 3.3.86, theorems 3.3.6 and 3.3.14 and propositional calculus}\} \\
& \mathbf{S}(\neg\mathbf{S}(\neg P_1; \mathbf{S}(\mathbf{true})) \vdash (\mathbf{S}(P2)[0.c'/c'] \wedge c', x' = c, x); \\
& \quad ((x', x.out'_c, x.in' = ((x.in_{0.c-1}, x) \lhd c < 0.c \rhd (x, x.out_c)), x.in) \wedge \\
& \quad\quad \left\{x.out'_i = x.in_{i-1} \mid c < i < 0.c\right\} \wedge c' = 0.c); \mathbf{end}\ 0.c) \\
= & \{\text{Assumption } (P \text{ is } \mathbf{SH3}), \text{ definition 2.2.3 and one-point rule } (P \text{ does not mention } x, \\
& \quad \text{it trivially keeps } \mathbf{S2} \text{ by performing an implicit } x.out' = x.out)\} \\
& \mathbf{S}(P_1 \vdash \exists c_0, 0.c_0 \bullet \mathbf{S}(P_2)[0.c_0/c'] \wedge c_0 = c \wedge \\
& \quad ((x', x.out'_{c_0}, x.in' = ((x.in_{0.c_0-1}, x) \lhd c_0 < 0.c_0 \rhd (x, x.out_{c_0}), x.in)) \wedge
\end{array}$$

$$\left\{x.out'_i = x.in_{i-1} \mid c_0 < i < 0.c_0\right\}) \wedge c' = 0.c_0)$$

= {One-point rule}

$$\mathbf{S}(P_1 \vdash \mathbf{S}(P_2) \wedge (x', x.out'_c, x.in' = ((x.in_{c'-1}, x) \lhd c < c' \rhd (x, x.out_c)), x.in) \wedge \left\{x.out'_i = x.in_{i-1} \mid c < i < c'\right\})$$

= {Definition 3.3.83}

$$\mathbf{S}(P_1 \vdash P_2 \wedge E(x, c'))$$

= {Definition 3.3.82}

$$P_{[x]}$$ □

Law 3.3.87 Provided that P is **S**-healhty and that $P = P \wedge c' = c$ we have:

$$P_{[x]} = P_{+x}$$

Proof.

$P_{[x]}$

= {Assumption and definition 3.3.82}

$$\mathbf{S}(P_1 \vdash P_2 \wedge c' = c \wedge ((x', x.out'_c, x.in' = ((x.in_{c'-1}, x) \lhd c < c' \rhd (x, x.out_c)), x.in) \wedge \left\{x.out'_i = x.in_{i-1} \mid c < i < c'\right\} \wedge \{I\!I_{\{x.out_i\}} \mid i < c\}))$$

= {Propositional calculus}

$$\mathbf{S}(P_1 \vdash P_2 \wedge c' = c \wedge (x', x.out'_c, x.in' = x, x.out_c, x.in) \wedge \{I\!I_{\{x.out_i\}} \mid i < c\})$$

= {P is **S**, propositional calculus and definition 2.3.46}

P_{+x} □

Law 3.3.88 $I\!I_{[x]} = I\!I_{+x} = I\!I$

Proof.

Straightforward from law 3.3.87. □

Law 3.3.89 $(P \lhd b \rhd Q)_{[x]} = (P_{[x]} \lhd b \rhd Q_{[x]})$

Proof.

$(P \lhd b \rhd Q)_{[x]}$

= {Theorem 3.3.21, definition 3.3.82}

$$\mathbf{S}(P_1 \lhd b \rhd Q_1 \vdash (P_2 \lhd b \rhd Q_2) \wedge E(x, c'))$$

= {Definition 2.2.1, propositional calculus}

$$\mathbf{S}(P_1 \lhd b \rhd Q_1 \vdash (b \wedge P_2 \wedge E(x, c')) \vee (\neg b \wedge Q_2 \wedge E(x, c')))$$

= {Definition 2.2.1}

$$\mathbf{S}(P_1 \lhd b \rhd Q_1 \vdash (P_2 \wedge E(x, c')) \lhd b \rhd (Q_2 \wedge E(x, c')))$$

= {Theorem 3.3.21, definition 3.3.82}

$P_{[x]} \lhd b \rhd Q_{[x]}$ □

Law 3.3.90 Provided P and Q are **S**-healthy we have:

$$(P; Q)_{[x]} \sqsubseteq (P_{[x]}; Q_{[x]})$$

We begin by proving the following lemma:

Lemma B.0.3.

$$\begin{array}{l}(P; Q) \wedge E(x, c') = \\ (\exists v_0, c_0, x_0, x.in.s, x.out.s \bullet \\ \quad P[v_0, c_0/v', c'] \wedge ((x_0, x.out.s_c, x.in.s = ((x.in_{c_0-1}, x) \lhd c < c_0 \rhd (x, x.out_c)), x.in) \wedge \\ \quad \{x.out.s_i = x.in_{i-1} \mid c < i < c_0\} \wedge \left\{x.out.s'_i = x.out_i \mid i < c \vee i \geq c_0\right\}) \wedge \\ \quad Q[v_0, c_0/v, c]((x', x.out'_{c_0}, x.in' = ((x.in.s_{c'-1}, x_0) \lhd c_0 < c' \rhd (x_0, x.out.s_{c_0})), x.in.s) \wedge \\ \quad \left\{x.out'_i = x.in.s_{i-1} \mid c_0 < i < c'\right\} \wedge \left\{x.out'_i = x.out.s_i \mid i < c_0\right\}))\end{array}$$

provided P and Q are **S***-healthy.*

Proof.

By case split on the relationship between c, $0.c$ and c' (P and Q are **S1** $\rightarrow c \leq c_0 \leq c'$)

Case $c = c_0 = c'$

$$\begin{array}{l}\exists v_0, c_0, x_0, x.in.s, x.out.s \bullet \\ \quad P[v_0, c_0/v', c'] \wedge ((x_0, x.out.s_c, x.in.s = ((x.in_{c_0-1}, x) \lhd c < c_0 \rhd (x, x.out_c)), x.in) \wedge \\ \quad \{x.out.s_i = x.in_{i-1} \mid c < i < c_0\} \wedge \{x.out.s_i = x.out_i \mid i < c \vee i \geq c_0\}) \wedge \\ \quad Q[v_0, c_0/v, c]((x', x.out'_{c_0}, x.in' = ((x.in.s_{c'-1}, x_0) \lhd c_0 < c' \rhd (x_0, x.out.s_{c_0})), x.in.s) \wedge \\ \quad \left\{x.out'_i = x.in.s_{i-1} \mid c_0 < i < c'\right\} \wedge \left\{x.out'_i = x.out.s_i \mid i < c_0\right\})\end{array}$$

= {$c = c_0 = c'$, propositional calculus}

$$\begin{array}{l}\exists v_0, c_0, x_0, x.in.s, x.out.s \bullet \\ \quad P[v_0, c_0/v', c'] \wedge ((x_0, x.out.s_c, x.in.s = x, x.out_c, x.in) \wedge x.out.s' = x.out) \wedge \\ \quad Q[v_0, c_0/v, c] \wedge ((x', x.in' = x_0, x.in.s) \wedge x.out' = x.out.s)\end{array}$$

= {One-point rule, quantifier contract scope}

$$\begin{array}{l}(\exists v_0, c_0 \bullet P[v_0, c_0/v', c'] \wedge Q[v_0, c_0/v, c]) \wedge ((x', x.in' = x, x.in) \wedge x.out' = x.out)\end{array}$$

= {$c = c_0 \wedge c_0 = c'$, predicate calculus}

$$\begin{array}{l}(\exists v_0, c_0 \bullet P[v_0, c_0/v', c'] \wedge Q[v_0, c_0/v, c]) \wedge \\ \quad ((x', x.out'_c, x.in' = ((x.in_{c'-1}, x) \lhd c < c' \rhd (x, x.out_c)), x.in) \wedge \\ \quad \left\{x.out'_i = x.in_{i-1} \mid c < i < c'\right\} \wedge \{I\!I_{\{x.out_i\}} \mid i < c\})\end{array}$$

= {Definitions 2.2.3, and 3.3.83}

$(P; Q) \wedge E(x, c')$

Case $c < c_0 \wedge c_0 = c'$

$\exists v_0, c_0, x_0, x.in.s, x.out.s \bullet$
$P[v_0, c_0/v', c'] \wedge ((x_0, x.out.s_c, x.in.s = ((x.in_{c_0-1}, x) \lhd c < c_0 \rhd (x, x.out_c)), x.in) \wedge$
$\{x.out.s_i = x.in_{i-1} \mid c < i < c_0\} \wedge \{x.out.s_i = x.out_i \mid i < c \vee i \geq c_0\}) \wedge$
$Q[v_0, c_0/v, c] \wedge ((x', x.out'_{c_0}, x.in' = ((x.in.s_{c'-1}, x_0) \lhd c_0 < c' \rhd (x_0, x.out.s_{c_0})), x.in.s) \wedge$
$\left\{x.out'_i = x.in.s_{i-1} \mid c_0 < i < c'\right\} \wedge \left\{x.out'_i = x.out.s_i \mid i < c_0\right\})$

= {$c < c_0 \wedge c_0 = c'$, propositional calculus}

$(\exists v_0, c_0, x_0, x.in.s, x.out.s \bullet$
$P[v_0, c_0/v', c'] \wedge (x_0, x.out.s_c, x.in.s = x.in_{c'-1}, x, x.in) \wedge$
$\{x.out.s_i = x.in_{i-1} \mid c < i < c_0\} \wedge \{x.out.s_i = x.out_i \mid i < c \vee i \geq c_0\}) \wedge$
$Q[v_0, c_0/v, c] \wedge ((x', x.out'_{c_0}, x.in' = (x_0, x.out.s_{c_0}, x.in.s) \wedge x.out' = x.out.s))$

= {One-point rule, quantifier contract scope}

$(\exists v_0, c_0 \bullet P[v_0, c_0/v', c'] \wedge Q[v_0, c_0/v, c]) \wedge (x', x.in' = x.in_{c'-1}, x.in) \wedge$
$x.out'_c = x \wedge \left\{x.out'_i = x.in_{i-1} \mid c < i < c'\right\} \wedge \left\{x.out'_i = x.out_i \mid i < c\right\}$

= {$c < c_0 = c'$, predicate calculus}

$(\exists v_0, c_0 \bullet P[v_0, c_0/v', c'] \wedge Q[v_0, c_0/v, c]) \wedge$
$((x', x.out'_c, x.in' = ((x.in_{c'-1}, x) \lhd c < c' \rhd (x, x.out_c)), x.in) \wedge$
$\left\{x.out'_i = x.in_{i-1} \mid c < i < c'\right\} \wedge \{I\!I_{\{x.out_i\}} \mid i < c\})$

= {Definitions 2.2.3 and 3.3.83}

$(P; Q) \wedge E(x, c')$

Case $c = c_0 \wedge c_0 < c'$

$\exists v_0, c_0, x_0, x.in.s, x.out.s \bullet$
$P[v_0, c_0/v', c'] \wedge ((x_0, x.out.s_c, x.in.s = ((x.in_{c_0-1}, x) \lhd c < c_0 \rhd (x, x.out_c)), x.in) \wedge$
$\{x.out.s_i = x.in_{i-1} \mid c < i < c_0\} \wedge \{x.out.s_i = x.out_i \mid i < c \vee i \geq c_0\}) \wedge$
$Q[v_0, c_0/v, c] \wedge ((x', x.out'_{c_0}, x.in' = ((x.in.s_{c'-1}, x_0) \lhd c_0 < c' \rhd (x_0, x.out.s_{c_0})), x.in.s) \wedge$
$\left\{x.out'_i = x.in.s_{i-1} \mid c_0 < i < c'\right\} \wedge \left\{x.out'_i = x.out.s_i \mid i < c_0\right\})$

= {$c = c_0 \wedge c_0 < c'$, propositional calculus}

$(\exists v_0, c_0, x_0, x.in.s, x.out.s \bullet$
$P[v_0, c_0/v', c'] \wedge (x_0, x.in.s = x, x.in) \wedge x.out.s = x.out \wedge$
$Q[v_0, c_0/v, c] \wedge (x', x.out'_{c_0}, x.in' = x.in.s_{c'-1}, x_0, x.in.s) \wedge$
$\left\{x.out'_i = x.in.s_{i-1} \mid c < i < c'\right\} \wedge \left\{x.out'_i = x.out.s_i \mid i < c\right\}$

= {One-point rule, quantifier contract scope}

$(\exists v_0, c_0 \bullet P[v_0, c_0/v', c'] \wedge Q[v_0, c_0/v, c]) \wedge$

$(x', x.out'_c, x.in' = x.in_{c'-1}, x, x.in) \wedge$

$\left\{x.out'_i = x.in_{i-1} \mid c < i < c'\right\} \wedge \left\{x.out'_i = x.out_i \mid i < c\right\}$

= {$c = c_0 \wedge c_0 < c'$, predicate calculus}

$(\exists v_0, c_0 \bullet P[v_0, c_0/v', c'] \wedge Q[v_0, c_0/v, c]) \wedge$

$((x', x.out'_c, x.in' = ((x.in_{c'-1}, x) \lhd c < c' \rhd (x, x.out_c)), x.in) \wedge$

$\left\{x.out'_i = x.in_{i-1} \mid c < i < c'\right\} \wedge \{I\!I_{\{x.out_i\}} \mid i < c\})$

= {Definitions 2.2.3 and 3.3.83}

$(P; Q) \wedge E(x, c')$

Case $c < c_0 < c'$

$\exists v_0, c_0, x_0, x.in.s, x.out.s \bullet$

$P[v_0, c_0/v', c'] \wedge ((x_0, x.out.s_c, x.in.s = ((x.in_{c_0-1}, x) \lhd c < c_0 \rhd (x, x.out_c)), x.in) \wedge$

$\{x.out.s_i = x.in_{i-1} \mid c < i < c_0\} \wedge \{x.out.s_i = x.out_i \mid i < c \vee i \geq c_0\}) \wedge$

$Q[v_0, c_0/v, c] \wedge ((x', x.out'_{c_0}, x.in' = ((x.in.s_{c'-1}, x_0) \lhd c_0 < c' \rhd (x_0, x.out.s_{c_0})), x.in.s) \wedge$

$\left\{x.out'_i = x.in.s_{i-1} \mid c_0 < i < c'\right\} \wedge \left\{x.out'_i = x.out.s_i \mid i < c_0\right\})$

= {$c < c_0 < c'$, propositional calculus}

$(\exists v_0, c_0, x_0, x.in.s, x.out.s \bullet$

$P[v_0, c_0/v', c'] \wedge (x_0, x.out.s_c, x.in.s = x.in_{c_0-1}, x, x.in) \wedge$

$\{x.out.s_i = x.in_{i-1} \mid c < i < c_0\} \wedge \{x.out.s_i = x.out_i \mid i < c \vee i \geq c_0\}) \wedge$

$Q[v_0, c_0/v, c] \wedge (x', x.out'_{c_0}, x.in' = x.in.s_{c'-1}, x_0, x.in.s) \wedge$

$\left\{x.out'_i = x.in.s_{i-1} \mid c_0 < i < c'\right\} \wedge \left\{x.out'_i = x.out.s_i \mid i < c_0\right\}$

= {One-point rule, quantifier contract scope, predicate calculus}

$(\exists v_0, c_0 \bullet P[v_0, c_0/v', c'] \wedge Q[v_0, c_0/v, c]) \wedge$

$(x', x.out'_{c_0}, x.in' = x.in_{c'-1}, x.in_{c_0-1}, x.in) \wedge$

$\left\{x.out'_i = x.in_{i-1} \mid c_0 < i < c'\right\} \wedge$

$\left\{x.out'_i = x.in_{i-1} \mid c < i < c_0\right\} \wedge (x.out'_c = x) \wedge \left\{x.out'_i = x.out_i \mid i < c\right\}$

= {$c < c_0 = c'$, predicate calculus}

$(\exists v_0, c_0 \bullet P[v_0, c_0/v', c'] \wedge Q[v_0, c_0/v, c]) \wedge$

$((x', x.out'_c, x.in' = ((x.in_{c'-1}, x) \lhd c < c' \rhd (x, x.out_c)), x.in) \wedge$

$\left\{x.out'_i = x.in_{i-1} \mid c < i < c'\right\} \wedge \{I\!I_{\{x.out_i\}} \mid i < c\})$

= {Definitions 2.2.3 and 3.3.83}

$(P; Q) \wedge E(x, c')$ □

With these result we can now prove law 3.3.90:

Proof.

Observation: For any variables P_1, Q, R, S and P_2 we have that

$$
\begin{array}{ll}
& (P_1 \wedge \neg(Q \wedge R); S \vdash P_2) \\
= & \{\text{Propositional calculus}\} \\
& (P_1 \wedge (\neg Q \vee \neg R); S \vdash P_2) \\
\sqsupseteq & \{\text{Removing the alternative of } \neg R \text{ strengthens the precondition and leads to refinement}\} \\
& (P_1 \wedge (\neg Q); S \vdash P_2)
\end{array}
$$

$$
\begin{array}{ll}
& P_{[x]}; Q_{[x]} \\
= & \{\text{Theorem 3.3.6 and definition 3.3.82}\} \\
& \mathbf{S}(\neg(\mathbf{S}(\neg P_1); \mathbf{S}(\mathbf{true})) \wedge \neg(\mathbf{S}(P_2 \wedge E(x, c')); \mathbf{S}(\neg Q_1)) \vdash \mathbf{S}(P_2 \wedge E(x, c')); \mathbf{S}(Q_2 \wedge E(x, c'))) \\
\sqsupseteq & \{\text{Observation above } (R = E(x, c')) \text{ and theorem 2.3.6}\} \\
& \mathbf{S}(\neg(\mathbf{S}(\neg P_1); \mathbf{S}(\mathbf{true})) \wedge \neg(\mathbf{S}(P_2); \mathbf{S}(\neg Q_1)) \vdash \\
& \quad (P_2 \wedge ((x', x.out'_c, x.in' = ((x.in_{c'-1}, x) \lhd c < c' \rhd (x, x.out_c)), x.in) \wedge \\
& \qquad \left\{x.out'_i = x.in_{i-1} \mid c < i < c'\right\} \wedge \{I\!I_{\{x.out_i\}} \mid i < c\})); \\
& \quad (Q_2((x', x.out'_c, x.in' = ((x.in_{c'-1}, x) \lhd c < c' \rhd (x, x.out_c)), x.in) \wedge \\
& \qquad \left\{x.out'_i = x.in_{i-1} \mid c < i < c'\right\} \wedge \{I\!I_{\{x.out_i\}} \mid i < c\}))) \\
= & \{\text{Definitions 2.2.3 and 2.2.7}\} \\
& \mathbf{S}(\neg(\mathbf{S}(\neg P_1); \mathbf{S}(\mathbf{true})) \wedge \neg(\mathbf{S}(P_2); \mathbf{S}(\neg Q_1)) \vdash (\exists v_0, c_0, x_0, x.in.s, x.out.s\bullet \\
& \quad P_2[v_0, c_0/v', c'] \wedge ((x_0, x.out.s_c, x.in.s = ((x.in_{c_0-1}, x) \lhd c < c_0 \rhd (x, x.out_c)), x.in) \wedge \\
& \qquad \{x.out.s_i = x.in_{i-1} \mid c < i < c_0\} \wedge \{x.out.s_i = x.out_i \mid i < c \vee i \geq c_0\}) \wedge \\
& \quad Q_2[v_0, c_0/v, c]((x', x.out'_{c_0}, x.in' = ((x.in.s_{c'-1}, x_0) \lhd c_0 < c' \rhd (x_0, x.out.s_{c_0})), x.in.s) \wedge \\
& \qquad \left\{x.out'_i = x.in.s_{i-1} \mid c_0 < i < c'\right\} \wedge \{x.out_i = x.out.s_i \mid i < c_0\}))) \\
= & \{\text{Lemma B.0.3}\} \\
& \mathbf{S}(\neg(\mathbf{S}(\neg P_1); \mathbf{S}(\mathbf{true})) \wedge \neg(\mathbf{S}(P_2); \mathbf{S}(\neg Q_1)) \vdash (\mathbf{S}(P_2); \mathbf{S}(Q_2)) \wedge E(x, c')) \\
= & \{\text{Theorem 3.3.6 and definition 3.3.82}\} \\
& (P; Q)_{[x]}
\end{array}
$$

□

Law 3.3.91 Provided P is $\mathbf{SH3} \circ \mathbf{S}$-healthy we have:

$$(b * P)_{[x]} \sqsubseteq b * P_{[x]}$$

Proof.

Let

$$F(X) = (P; X) \lhd b \rhd \mathbb{I}$$
$$G(X) = (P_{[x]}; X) \lhd b \rhd \mathbb{I}$$

We first show that

$$F^i(\bot)_{[x]} \sqsubseteq G^i(\bot)$$

For $i = 0$ we have:

$$\begin{aligned}
& F^0(\bot)_{[x]} \\
= & \{\text{Definition of } F^0 \text{ and predicate calculus}\} \\
& \bot \\
= & \{\text{Definition of } G^0\} \\
& G^0(\bot)
\end{aligned}$$

□

For $i = n + 1$ we have:

$$\begin{aligned}
& F^{n+1}(\bot)_{[x]} \\
= & \{\text{Definition of } F^{i+1}\} \\
& ((P; F^n(\bot)) \lhd b \rhd \mathbb{I})_{[x]} \\
\sqsubseteq & \{\text{Laws 3.3.89, 3.3.88, 3.3.90}\} \\
& (P_{[x]}; F^n(\bot)_{[x]}) \lhd b \rhd \mathbb{I} \\
\sqsubseteq & \{\text{Inductive hypothesis}\} \\
& (P_{[x]}; G^n(\bot)) \lhd b \rhd \mathbb{I} \\
= & \{\text{Definition of } G^{n+1}\} \\
& G^{n+1}(\bot)
\end{aligned}$$

With this result we now prove that $(b * P)_{[x]} \sqsubseteq b * P_{[x]}$:

$$\begin{aligned}
& (b * P)_{[x]} \\
= & \{\text{Law 3.3.85}\} \\
& ((b * P); U0(c))_{+x,c}; E_D(x)
\end{aligned}$$

= {Law 2.3.139 (all operators in the language are continuous)}

$$\left(\left(\bigsqcup_i F^i(\mathbf{true})\right); U0(c)\right)_{+x,c}; E_D(x)$$

= {Law 3.3.11, alphabet extension distributes over lub}

$$\bigsqcup_i \left((F^i(\mathbf{true}); U0(c))_{+x,c}; E_D(x)\right)$$

= {Law 3.3.85}

$$\bigsqcup_i F^i(\bot)_{[x]}$$

$\sqsubseteq$ {Observation above}

$$\bigsqcup_i G^i(\bot)$$

= {Law 2.3.139 and definition of G}

$$b * P_{[x]}$$

Law 3.3.92 If P is **SH3** and **S**-healthy, it does not perform `sync` events and neither P nor **S** mention x then we have:

$$P; \mathtt{var}\ x; Q = \mathtt{var}\ x; P_{[x]}; Q$$

Proof.

$\mathtt{var}\ x; P_{[x]}; Q$

= {Definition 3.3.64, assumption: P does not perform **sync** actions ($c' = c$), law 3.3.87}

$\mathbf{S}(\mathbf{true} \vdash \mathbf{var}\ x); \mathbf{S}(P_1 \vdash P_2 \wedge (x', x.out', x.in' = x, x.out, x.in)); Q$

= {Theorem 3.3.6, propositional calculus}

$\mathbf{S}(\neg(\mathbf{S}(\mathbf{var}\ x); \mathbf{S}(\neg P_1)) \vdash \mathbf{S}(\mathbf{var}\ x); \mathbf{S}(P_2 \wedge x' = x)); Q$

= {Theorem 3.3.68, definition 2.2.18}

$\mathbf{S}(\neg(\exists x \bullet \mathbf{S}(\neg P_1)) \vdash \exists x \bullet \mathbf{S}(P_2 \wedge x' = x)); Q$

= {Assumption: P and **S** do not mention x, superfluous quantifier, one point rule}

$\mathbf{S}(\neg(\mathbf{S}(\neg P_1)) \vdash \mathbf{S}(P_2)); Q$

= {Assumption (P is **S**-healthy), definitions 2.3.1 and 2.2.3}

$\exists x_0, v_0 \bullet P[v_0, x_0/v', x']; Q[v_0, x_0/v, x]$

= {P does not mention x', quantifier contract scope, definition 2.2.3}

$P; \exists x_0 \bullet Q[x_0/x]$

= {Change variable name, theorem 3.3.66}

$P; \mathtt{var}\ x; Q$ □

Law 3.3.93 If P and Q are **SH3** $\circ$ **S**-healthy, P does not perform any `sync` events and neither P

nor **S** mention x' then we have:

$$Q; \mathtt{end}\ x; P = Q; P_{[x]}; \mathtt{end}\ x$$

Proof.

$$\begin{aligned}
& Q; P_{[x]}; \mathtt{end}\ x \\
= & \ \{\text{Assumptions, law 3.3.87 and definition 3.3.65}\} \\
& Q; \mathbf{S}(P_1 \vdash P_2 \wedge x' = x); \mathbf{S}(\mathbf{true} \vdash \mathbf{end}\ x) \\
= & \ \{\text{Theorem 3.3.6 and propositional calculus}\} \\
& Q; \mathbf{S}(\neg \mathbf{S}(\neg P_1) \vdash \mathbf{S}(P_2 \wedge x' = x); \mathbf{S}(\mathbf{end}\ x)) \\
= & \ \{\text{Theorem 3.3.69, definition 2.2.19}\} \\
& Q; \mathbf{S}(\neg \mathbf{S}(\neg P_1) \vdash \exists x' \bullet \mathbf{S}(P_2 \wedge x' = x)) \\
= & \ \{\text{Assumption: neither } P \text{ nor } \mathbf{S} \text{ mention } x', \text{propositional calculus, one-point rule}\} \\
& Q; \mathbf{S}(\neg \mathbf{S}(\neg P_1) \vdash \mathbf{S}(P_2)) \\
= & \ \{\text{Assumption } (P \text{ is } \mathbf{S}\text{-healthy}), \text{definitions 2.3.1 and 2.2.3}\} \\
& \exists v_0, x_0 \bullet Q[v_0, x_0/v', x']; P[v_0, x_0/v, x] \\
= & \ \{P \text{ does not mention } X \text{ or } x', \text{quantifier contract scope, definition 2.2.3}\} \\
& \exists x_0 \bullet Q[x_0/x']; P \\
= & \ \{\text{Change variable name, theorem 3.3.67}\} \\
& Q; \mathtt{end}\ x; P
\end{aligned}$$

□

Law 3.3.94 If P is **SH3** and **S**, it does perform at least one `sync` event and neither P nor **S** mention x then we have:

$$P; \mathtt{var}\ x; Q \sqsubseteq \mathtt{var}\ x; P_{[x]}; Q$$

Proof.

$$\begin{aligned}
& \mathtt{var}\ x; P_{[x]}; Q \\
= & \ \{\text{Definitions 3.3.64 and 3.3.82}\} \\
& (\mathbf{true} \vdash \mathbf{var}\ x); (P_1 \vdash P_2 \wedge E(x, c')); Q \\
= & \ \{\text{Theorem 3.3.6, propositional calculus}\} \\
& \mathbf{S}(\neg(\mathbf{S}(\mathbf{var}\ x); \mathbf{S}(\neg P_1)) \vdash \mathbf{S}(\mathbf{var}\ x); \mathbf{S}(P_2 \wedge E(x, c'))); Q \\
= & \ \{P \text{ performs at least one } \mathbf{sync} \rightarrow c' > c, \text{definition of } E(x, c') \text{ and propositional calculus}\} \\
& \mathbf{S}(\neg(\mathbf{S}(\mathbf{var}\ x); \mathbf{S}(\neg P_1)) \vdash \mathbf{S}(\mathbf{var}\ x); \mathbf{S}(P_2 \wedge \\
& \qquad (x', x.out'_c, x.in' = x.in_{c'-1}, x, x.in) \wedge \left\{x.out'_i = x.in_{i-1} \mid c < i < c'\right\} \wedge \{I\!I_{\{x.out_i\}} \mid i < c\})); Q \\
= & \ \{\text{Theorem 3.3.68, definition 2.2.18}\}
\end{aligned}$$

$$\begin{array}{l}
\quad \mathbf{S}(\neg(\exists x \bullet \mathbf{S}(\neg P_1)) \vdash \exists x \bullet \mathbf{S}(P_2 \wedge (x', x.out'_c, x.in' = x.in_{c'-1}, x, x.in) \wedge \\
\qquad \left\{x.out'_i = x.in_{i-1} \mid c < i < c'\right\} \wedge \{I\!I_{\{x.out_i\}} \mid i < c\})); Q \\
= \{\text{One-point rule, } P_2 \text{ is } \mathbf{S3} \text{ and trivially satisfies } \forall i < c \bullet x.out'_i = x.out_i\} \\
\quad \mathbf{S}(\neg(\mathbf{S}(\neg P_1)) \vdash \mathbf{S}(P_2 \wedge x.out'_c = x \wedge \left\{x.out'_i = x.in_{i-1} \mid c < i < c'\right\})); Q \\
\sqsupseteq \{P_2 \wedge Q \Rightarrow P, \text{ monotonicity of designs, theorem 2.3.6}\} \\
\quad \mathbf{S}(\neg(\mathbf{S}(\neg P_1)) \vdash \mathbf{S}(P_2)); Q \\
= \{\text{Assumption } (P \text{ is } \mathbf{S}\text{-healthy}), \text{ definitions 2.3.1 and 2.2.3}\} \\
\quad \exists x_0, v_0 \bullet P[v_0, x_0/v', x']; Q[v_0, x_0/v, x] \\
= \{P \text{ does not mention } x', \text{ quantifier contract scope, definition 2.2.3}\} \\
\quad P; \exists x_0 \bullet Q[x_0/x] \\
= \{\text{Change variable name, theorem 3.3.66}\} \\
\quad P; \mathtt{var}\ x; Q
\end{array}$$

□

Law 3.3.95 If P and Q are $\mathbf{SH3} \circ \mathbf{S}$-healthy, P does perform at least one `sync` event and neither P nor $\mathbf{S}$ mention x' then we have:

$$Q; \mathsf{end}\ x; P \sqsubseteq Q; P_{[x]}; \mathsf{end}\ x$$

Proof.

$$\begin{array}{l}
\quad Q; P_{[x]}; \mathsf{end}\ x \\
= \{\text{Definitions 3.3.82 and 3.3.65, assumption and propositional calculus}\} \\
\quad Q; \mathbf{S}(P_1 \vdash P_2 \wedge (x', x.out'_c, x.in' = x.in_{c'-1}, x, x.in) \wedge \\
\qquad \left\{x.out'_i = x.in_{i-1} \mid c < i < c'\right\} \wedge \{I\!I_{\{x.out_i\}} \mid i < c\}); \mathbf{S}(\mathbf{true} \vdash \mathbf{end}\ x) \\
= \{\text{Theorem 3.3.6, propositional calculus}\} \\
\quad Q; \mathbf{S}(\neg\mathbf{S}(\neg P_1) \vdash \mathbf{S}(P_2 \wedge (x', x.out'_c, x.in' = x.in_{c'-1}, x, x.in) \wedge \\
\qquad \left\{x.out'_i = x.in_{i-1} \mid c < i < c'\right\} \wedge \{I\!I_{\{x.out_i\}} \mid i < c\}); \mathbf{S}(\mathbf{end}\ x)) \\
= \{\text{Theorem 3.3.69 and definition 2.2.19}\} \\
\quad Q; \mathbf{S}(\neg\mathbf{S}(\neg P_1) \vdash \exists x' \bullet \mathbf{S}(P_2 \wedge (x', x.out'_c, x.in' = x.in_{c'-1}, x, x.in) \wedge \\
\qquad \left\{x.out'_i = x.in_{i-1} \mid c < i < c'\right\} \wedge \{I\!I_{\{x.out_i\}} \mid i < c\})) \\
= \{\text{One-point rule, } P_2 \text{ is } \mathbf{S3} \text{ and trivially satisfies } \forall i < c \bullet x.out'_i = x.out_i\} \\
\quad Q; \mathbf{S}(\neg\mathbf{S}(\neg P_1) \vdash \mathbf{S}(P_2 \wedge x.out'_c = x \wedge \left\{x.out'_i = x.in_{i-1} \mid c < i < c'\right\})) \\
\sqsupseteq \{P_2 \wedge Q \Rightarrow P_2, \text{ definition of design refinement}\} \\
\quad Q; \mathbf{S}(\neg\mathbf{S}(\neg P_1) \vdash \mathbf{S}(P_2)) \\
= \{\text{Assumption } (P \text{ is } \mathbf{S}\text{-healthy}), \text{ definitions 2.3.1 and 2.2.3}\} \\
\quad \exists v_0, x_0 \bullet Q[v_0, x_0/v', x']; P[v_0, x_0/v, x]
\end{array}$$

= {P does not mention X or x', quantifier contract scope, definition 2.2.3}

$\exists x_0 \bullet Q[x_0/x']; P$

= {Change variable name, theorem 3.3.67}

Q; end x; P □

Law 3.3.97 $b * P = (P; b * P) \lhd b \rhd \text{II}$

Proof.

Straightforward from definition 3.3.96 and law 2.3.80. □

Law 3.3.98 $(\neg b)^\top_{\text{S}}; b * P = (\neg b)^\top_{\text{S}}$

Proof.

$(\neg b)^\top_{\text{S}}; b * P$

= {Definition 3.3.34, laws 3.3.33, 3.3.9 and 3.3.7}

$(b * P) \lhd \neg b \rhd \top$

= {Law 3.3.97 and 3.3.25}

$\text{II} \lhd \neg b \rhd \top$

= {Definition 3.3.34}

$(\neg b)^\top_{\text{S}}$ □

Law 3.3.99 $(\neg b)^\bot_{\text{S}}; b * P = (\neg b)^\bot_{\text{S}}$

Proof.

Similar to proof of law 3.3.98. □

Law 3.3.100 $b^\top_{\text{S}}; b * P = b^\top_{\text{S}}; P; (b * P)$

Proof.

$b^\top_{\text{S}}; b * P$

= {Law 3.3.97}

$b^\top_{\text{S}}; ((P; b * P) \lhd b \rhd \text{II})$

= {Law 3.3.58}

$b^\top_{\text{S}}; (P; b * P)$ □

Law 3.3.101 $b * P = (b * P); (\neg b)^\bot_{\text{S}}$

Proof.

Let $F(X) =_{df} (P; X) \lhd b \rhd \text{Ⅱ}$, we are interested in proving that

$$\forall n \in \mathbb{N} \bullet F^n(\bot) = F^n(\bot); (\neg b)^{\bot}_{\mathsf{S}}$$

We conduct the proof by induction on n:
Base case ($n = 0$):

$$\begin{aligned}
& F^0(\bot) \\
= & \{\text{Definition } F^x\} \\
& \bot \\
= & \{\text{Law 3.3.8 and definition of } F^x\} \\
& F^0(\bot); (\neg b)^{\bot}_{\mathsf{S}}
\end{aligned}$$

Induction hypothesis: $F^n(\bot) = F^n(\bot); (\neg b)^{\bot}_{\mathsf{S}}$
Step case ($n + 1$):

$$\begin{aligned}
& F^{n+1} \\
= & \{\text{Definition } F^x \text{ and definition of } F\} \\
& (P; F^n(\bot)) \lhd b \rhd \text{Ⅱ} \\
= & \{\text{Inductive hypothesis, law 3.3.57}\} \\
& (b^{\top}_{\mathsf{S}}; P; F^n(\bot); (\neg b)^{\bot}_{\mathsf{S}}) \lhd b \rhd ((\neg b)^{\top}_{\mathsf{S}}; \text{Ⅱ}) \\
= & \{\text{Laws 3.3.52, 3.3.41, 3.3.57}\} \\
& ((b^{\top}_{\mathsf{S}}; P; F^n(\bot)) \lhd b \rhd (\neg b)^{\top}_{\mathsf{S}}); (\neg b)^{\bot}_{\mathsf{S}} \\
= & \{\text{Law 3.3.57 and definition of } F^x \text{ and } F\} \\
& F^{n+1}(\bot); (\neg b)^{\bot}_{\mathsf{S}}
\end{aligned}$$

With this result we can now prove that: $b * P = (b * P); (\neg b)^{\bot}_{\mathsf{S}}$

$$\begin{aligned}
& (b * P); (\neg b)^{\bot}_{\mathsf{S}} \\
= & \{\text{Definitions 2.3.82 and } F \text{ above, law 2.3.139 (all operators are continuous)}\} \\
& (\bigsqcap_i F^i(\mathbf{true})); (\neg b)^{\bot}_{\mathsf{S}} \\
= & \{\text{Law 2.3.135}\} \\
& \bigsqcap_i (F^i(\mathbf{true}); (\neg b)^{\bot}_{\mathsf{S}}) \\
= & \{\text{Observation above}\}
\end{aligned}$$

$$\bigsqcup_i F^i(\mathbf{true})$$

= {Law 2.3.139 and definition of F}

$b * P$ □

Law 3.3.102 Provided $c_{\mathrm{S}}^{\perp}; P = c_{\mathrm{S}}^{\perp}; P; c_{\mathrm{S}}^{\perp}$ we have:

$$c_{\mathrm{S}}^{\perp}; (b * P) = c_{\mathrm{S}}^{\perp}; (b * P); c_{\mathrm{S}}^{\perp}$$

Proof.

Let $F(X) =_{df} (P; X) \lhd b \rhd \text{Ⅱ}$, we are interested in proving that

$$\forall n \in \mathbb{N} \bullet (\neg c)_{\mathrm{S}}^{\perp}; F^n(\perp) = (\neg c)_{\mathrm{S}}^{\perp}; F^n(\perp); (\neg c)_{\mathrm{S}}^{\perp}$$

We conduct the proof by induction on n:
Base case ($n = 0$):

$(\neg c)_{\mathrm{S}}^{\perp}; F^0(\perp)$

= {Definition F^x}

$(\neg c)_{\mathrm{S}}^{\perp}; \perp$

= {Law 3.3.8 and definition of F^x}

$(\neg c)_{\mathrm{S}}^{\perp}; F^0(\perp); (\neg c)_{\mathrm{S}}^{\perp}$

Induction hypothesis: $(\neg c)_{\mathrm{S}}^{\perp}; F^n(\perp) = (\neg c)_{\mathrm{S}}^{\perp}; F^n(\perp); (\neg c)_{\mathrm{S}}^{\perp}$
Step case ($n + 1$):

$(\neg c)_{\mathrm{S}}^{\perp}; F^{n+1}$

= {Definition F^x and definition of F}

$(\neg c)_{\mathrm{S}}^{\perp}; ((P; F^n(\perp)) \lhd b \rhd \text{Ⅱ})$

= {Law 3.3.63 and inductive hypothesis}

$((\neg c)_{\mathrm{S}}^{\perp}; P; (\neg c)_{\mathrm{S}}^{\perp}; F^n(\perp); (\neg c)_{\mathrm{S}}^{\perp}) \lhd b \rhd ((\neg c)_{\mathrm{S}}^{\perp}; \text{Ⅱ})$

= {Laws 3.3.52, 3.3.49 and 3.3.33}

$(((\neg c)_{\mathrm{S}}^{\perp}; P; F^n(\perp)) \lhd b \rhd (\neg c)_{\mathrm{S}}^{\perp}); (\neg c)_{\mathrm{S}}^{\perp}$

= {Laws 3.3.52, assumption, 3.3.63}

$(\neg c)_{\mathrm{S}}^{\perp}; ((P; F^n(\perp)) \lhd b \rhd \text{Ⅱ}); (\neg c)_{\mathrm{S}}^{\perp}$

= {Definitions of F^x and F}

$(\neg c)_{\mathrm{S}}^{\perp}; F^{n+1}(\perp); (\neg c)_{\mathrm{S}}^{\perp}$

With this result we can now prove that: $(\neg c)^{\perp}_{\mathrm{S}}; (b * P) = (\neg c)^{\perp}_{\mathrm{S}}; (b * P); (\neg c)^{\perp}_{\mathrm{S}}$

$(\neg c)^{\perp}_{\mathrm{S}}; (b * P); (\neg c)^{\perp}_{\mathrm{S}}$
= {Definitions 2.3.82 and F above, law 2.3.139 (all operators are continuous)}
$(\neg c)^{\perp}_{\mathrm{S}}; (\bigsqcap_i F^i(\perp)); (\neg c)^{\perp}_{\mathrm{S}}$
= {Laws 2.3.135 and 2.3.136 (we can always express an assertion as a finite normal form)}
$\bigsqcap_i ((\neg c)^{\perp}_{\mathrm{S}}; F^i(\perp); (\neg b)^{\perp}_{\mathrm{S}})$
= {Observation above}
$\bigsqcap_i ((\neg c)^{\perp}_{\mathrm{S}}; F^i(\perp))$
= {Laws 2.3.136 and 2.3.139 and definition of F}
$(\neg c)^{\perp}_{\mathrm{S}}; (b * P)$ □

Law 3.3.103 $(b * P); Q = \mu X \bullet (P; X) \lhd b \rhd Q$

Proof.

Let

$$F(X) =_{df} (P; X) \lhd b \rhd \text{Ⅱ}$$

and

$$G(X) =_{df} (P; X) \lhd b \rhd Q$$

we are interested in proving that:

$$\forall n \in \mathbb{N} \bullet F^n(\perp); Q = G^n(\perp)$$

We conduct the proof by induction on n:
Base case ($n = 0$):

$F^0(\perp); Q$
= {Definition F^x}
$\perp; Q$
= {Law 3.3.8 and definition of G^x}

$G^0(\bot)$

Induction hypothesis: $F^n(\bot); Q = G^n(\bot)$
Step case $(n + 1)$:

$F^{n+1}; Q$
= {Definition F^x and definition of F}
$(P; F^n(\bot)) \lhd b \rhd \text{Ⅱ}); Q$
= {Laws 3.3.33}
$(P; F^n(\bot); Q) \lhd b \rhd Q$
= {Inductive hypothesis}
$(P; G^n(\bot)) \lhd b \rhd Q$
= {Definitions of G^x and G}
$G^{n+1}(\bot)$

With this result we can now prove that: $\mu X \bullet (P; X) \lhd b \rhd Q = (b * P); Q$

$\mu X \bullet (P; X) \lhd b \rhd Q$
= {Definition of G, law 2.3.139 (all operators are continuous)}
$\bigsqcup_i G^i(\bot)$
= {Result above}
$\bigsqcup_i (F^i(\bot); Q)$
= {Laws 2.3.136 and 2.3.139 and definition of F}
$(\mu X \bullet (P; X) \lhd b \rhd \text{Ⅱ}); Q$
= {Definition 3.3.96}
$(b * P); Q$ □

Law 3.3.104 $b * P; (b \vee q) * P = (b \vee q) * P$

Proof.

We follow the proof outline of the same law in the context of the design theory [Hoare and He 1998, Page 126]. We begin by defining $S(X) =_{df} (P; X) \lhd b \rhd (b \vee c) * P$. From law 3.3.103 we have:

$LHS = \mu S$

so we only need to show $(b \vee c) * P = \mu S$.

$$
\begin{array}{ll}
& (b \vee c) * P = (b \vee c) * P \\
\equiv & \{\text{Law 3.3.23}\} \\
& (b \vee c) * P = ((b \vee c) * P) \lhd b \rhd ((b \vee c) * P) \\
\equiv & \{\text{Laws 3.3.57 and 3.3.100}\} \\
& (b \vee c) * P = (P; (b \vee c) * P) \lhd b \rhd ((b \vee c) * P) \\
\Rightarrow & \{\text{Law 2.3.81}\} \\
& (b \vee c) * P \sqsupseteq \mu X \bullet (P; X) \lhd b \rhd ((b \vee c) * P) \\
\equiv & \{\text{Definition of } muS \text{ above}\} \\
& (b \vee c) * P \sqsupseteq \mu S
\end{array}
$$

To prove equality, we need to prove both directions in the refinement, so we have:

$$
\begin{array}{ll}
& \mu S = \mu S \\
\equiv & \{\text{Definition of } \mu S \text{ and law 2.3.80}\} \\
& \mu S = (P; \mu S) \lhd b \rhd (b \vee c) * P \\
\equiv & \{\text{Law 3.3.97}\} \\
& \mu S = (P; \mu S) \lhd b \rhd (P; (b \vee c) * P) \lhd (b \vee c) \rhd \text{II} \\
\equiv & \{\text{Result above } ((b \vee c) * P \sqsupseteq \mu S)\} \\
& \mu S \sqsupseteq (P; \mu S) \lhd b \rhd (P; \mu S) \lhd (b \vee c) \rhd \text{II} \\
\equiv & \{\text{Law 3.3.30}\} \\
& \mu S \sqsupseteq (P; \mu S) \lhd (b \vee c) \rhd \text{II} \\
\Rightarrow & \{\text{Law 2.3.81}\} \\
& \mu S \sqsupseteq \mu X \bullet (P; X) \lhd (b \vee c) \rhd \text{II} \\
\equiv & \{\text{Definition 3.3.96}\} \\
& \mu S \sqsupseteq (b \vee c) * P
\end{array}
$$

□

Law 3.3.105 Provided P is **S**-healthy and b does not mention x we have:

$$b * (\texttt{var}\ x; P; \texttt{end}\ x) \sqsubseteq \texttt{var}\ x; (b * P); \texttt{end}\ x$$

Proof.

$$
\begin{array}{ll}
& \texttt{var}\ x; b * P; \texttt{end}\ x = \texttt{var}\ x; b * P; \texttt{end}\ x \\
\equiv & \{\text{Definition 2.3.82, law 2.3.80}\} \\
& \texttt{var}\ x; b * P; \texttt{end}\ x = \texttt{var}\ x; ((P; b * P) \lhd b \rhd \text{II}); \texttt{end}\ x
\end{array}
$$

$\equiv$ {Laws 3.3.33 and 3.3.81}

$\texttt{var}\ x; b * P; \texttt{end}\ x \sqsupseteq (\texttt{var}\ x; P; (b * P); \texttt{end}\ x) \lhd b \rhd (\texttt{var}\ x; \Pi; \texttt{end}\ x)$

$\equiv$ {Laws 3.3.71, 3.3.9 and 3.3.70}

$\texttt{var}\ x; b * P; \texttt{end}\ x \sqsupseteq (\texttt{var}\ x; P; \texttt{end}\ x; \texttt{var}\ x; (b * P); \texttt{end}\ x) \lhd b \rhd \Pi$

$\Rightarrow$ {Law 2.3.81}

$\texttt{var}\ x; b * P; \texttt{end}\ x \sqsupseteq \mu X \bullet (\texttt{var}\ x; P; \texttt{end}\ x; X) \lhd b \rhd \Pi$

$\equiv$ {Definition 2.3.82}

$\texttt{var}\ x; b * P; \texttt{end}\ x \sqsupseteq b * (\texttt{var}\ x; P; \texttt{end}\ x)$ □

Law 3.3.108 $\mathbf{S}[\mathbf{0}](P) \parallel \mathbf{S}[\mathbf{1}](Q) = \mathbf{S}[\mathbf{1}](Q) \parallel \mathbf{S}[\mathbf{0}](P)$

Law 3.3.109 $\mathbf{S}[\mathbf{0}](P) \parallel (\mathbf{S}[\mathbf{1}](Q) \parallel \mathbf{S}[\mathbf{2}](R)) = (\mathbf{S}[\mathbf{0}](P) \parallel \mathbf{S}[\mathbf{1}](Q)) \parallel \mathbf{S}[\mathbf{2}](R)$

Law 3.3.110 $(\mathbf{S}[\mathbf{0}](P) \parallel \mathbf{S}[\mathbf{1}](\Pi_D)) = \mathbf{S}[\mathbf{0}] \circ \mathbf{S}[\mathbf{1}](P)$

Law 3.3.111 $\mathbf{S}[\mathbf{0}](P) \parallel \mathbf{S}[\mathbf{1}](\bot_D) = \mathbf{S}[\mathbf{0}] \circ \mathbf{S}[\mathbf{1}](\bot_D)$

Law 3.3.112 $\mathbf{S}(P \lhd b \rhd Q) \parallel \mathbf{S}[\mathbf{2}](R) = ((\mathbf{S}(P) \parallel \mathbf{S}[\mathbf{2}](R)) \lhd b \rhd (\mathbf{S}(Q) \parallel \mathbf{S}[\mathbf{2}](R)))$

Law 3.3.113 For any descending chain $S = \{S_n \mid n \in \mathbb{N}\}$ we have:

$$(\bigsqcap S) \parallel \mathbf{S}[\mathbf{2}](R) = \bigsqcap_n (S_n \parallel \mathbf{S}[\mathbf{2}](R))$$

Law 3.3.114 $\mathbf{S}[\mathbf{0}](x := e_1) \parallel \mathbf{S}[\mathbf{1}](y := e_2) = \mathbf{S}[\mathbf{0}] \circ \mathbf{S}[\mathbf{1}](x, y := e_1, e_2)$

The proofs of laws 3.3.108 to 3.3.114 are straightforward from definition 3.3.107 and laws 2.3.90 to 2.3.98 respectively.

Renaming the c, $x.in$ and $x.out$ variables in a **S**-healthy predicate has the effect of renaming the healthiness conditions as well. Fortunately, the merge function M, when applied to c and pointwise to $x.out$, restores the healthiness conditions to their state before the renaming. This fact will be useful in proving many properties about parallel by merge. The following laws capture this notion more precisely.

Law B.0.4.

$$\mathbf{S}[\mathbf{0}] \circ \mathbf{S}[\mathbf{1}](\textit{true}); M = \mathbf{S}(\textit{true})$$

Proof.

For the proof, we will show the case of **S1**. The proof of the other healthiness conditions follow a similar schema.

$\mathbf{S1}[\mathbf{0}] \circ \mathbf{S1}[\mathbf{1}](\mathbf{true}); M(c, 0.c, 1.c.c')$

= {Definitions 3.2.1 and 3.3.106}

$(c \leq 0.c' \wedge c \leq 1.c'); M(c, 0.c, 1.c.c')$

= {Case split}

$c = 0.c' \wedge (c \leq 0.c' \wedge c \leq 1.c'); M(c, 0.c, 1.c.c') \vee$

$c = 1.c' \wedge (c \leq 0.c' \wedge c \leq 1.c'); M(c, 0.c, 1.c.c') \vee$

$c \neq 0.c' \wedge c \neq 1.c' \wedge (c \leq 0.c' \wedge c \leq 1.c'); M(c, 0.c, 1.c.c')$

= {Predicate calculus, law 2.3.119}

$(c \leq 0.c' \wedge c \leq 1.c'); c' = 1.c \vee$

$(c \leq 0.c' \wedge c \leq 1.c'); c' = 0.c \vee$

$(c \leq 0.c' \wedge c \leq 1.c'); c' = 0.c \sqcap 1.c$

= {Definition 2.2.3, one-point rule, predicate calculus}

$\exists 0.c_0 \bullet c \leq 0.c_0 \wedge c \leq c' \vee$

$\exists 1.c_0 \bullet c \leq c' \wedge c \leq 1.c_0 \vee$

$(\exists 0.c_0 \bullet c \leq 0.c_0 \wedge c \leq c') \sqcap (\exists 1.c_0 \bullet c \leq c' \wedge c \leq 1.c_0)$

= {Propositional calculus (in all cases, a sufficient witness is just c}

$c \leq c'$

= {Definition 3.2.2}

$\mathbf{S1}(\mathbf{true})$

□

Law B.0.5. *Provided P does not mention the variables in the healthiness conditions we have:*

$$\mathbf{S}[\mathbf{0}] \circ \mathbf{S}[\mathbf{1}](P); M = \mathbf{S}(P)$$

Proof.

For the proof, we will show the case of **S1**. The proof of the other healthiness conditions follow a similar schema.

$\mathbf{S1}[\mathbf{0}] \circ \mathbf{S1}[\mathbf{1}](P); M(c, 0.c, 1.c.c')$

= {P does not mention the variables in $\mathbf{S1}[\mathbf{0}] \circ \mathbf{S1}[\mathbf{1}]$, predicate calculus}

$P \wedge \mathbf{S1}[\mathbf{0}] \circ \mathbf{S1}[\mathbf{1}](\mathbf{true}); M(c, 0.c, 1.c.c')$

= {Law B.0.4}

$P \wedge \mathbf{S1}(\mathbf{true})$

= {Definition 3.2.2}

$\mathbf{S}(P)$ □

Law B.0.6. *Provided P mentions the variables in the merge predicate we have:*

$$\mathbf{S}[\mathbf{0}] \circ \mathbf{S}[\mathbf{1}](P); M = \mathbf{S}(\mathbf{S}[\mathbf{0}] \circ \mathbf{S}[\mathbf{1}](P); M)$$

Proof.

For the proof, we will show the case of **S1**. The proof of the other healthiness conditions follow a similar schema.

$\mathbf{S1}[\mathbf{0}] \circ \mathbf{S1}[\mathbf{1}](P); M(c, 0.c, 1.c.c')$

= {Definition 3.2.1 and predicate calculus}

$(P \wedge c \leq 0.c' \wedge c \leq 1.c'); M(c, 0.c, 1.c.c')$

= {Case split}

$c = 0.c' \wedge (P \wedge c \leq 0.c' \wedge c \leq 1.c'); M(c, 0.c, 1.c.c') \vee$

$c = 1.c' \wedge (P \wedge c \leq 0.c' \wedge c \leq 1.c'); M(c, 0.c, 1.c.c') \vee$

$c \neq 0.c' \wedge c \neq 1.c' \wedge (P \wedge c \leq 0.c' \wedge c \leq 1.c'); M(c, 0.c, 1.c.c')$

= {Predicate calculus, law 2.3.119}

$(P \wedge c \leq 0.c' \wedge c \leq 1.c'); c' = 1.c \vee$

$(P \wedge c \leq 0.c' \wedge c \leq 1.c'); c' = 0.c \vee$

$(P \wedge c \leq 0.c' \wedge c \leq 1.c'); c' = 0.c \sqcap 1.c$

= {Definition 2.2.3}

$\exists 0.c_0, 1.c_0 \bullet (P[0.c_0, 1.c_0/0.c', 1.c'] \wedge c \leq 0.c' \wedge c \leq 1.c') \wedge c' = 1.c_0 \vee$

$\exists 0.c_0, 1.c_0 \bullet (P[0.c_0, 1.c_0/0.c', 1.c'] \wedge c \leq 0.c' \wedge c \leq 1.c') \wedge c' = 0.c_0 \vee$

$\exists 0.c_0, 1.c_0 \bullet (P[0.c_0, 1.c_0/0.c', 1.c'] \wedge c \leq 0.c' \wedge c \leq 1.c') \wedge c' = 0.c_0 \sqcap 1.c_0$

= {One-point rule, predicate calculus}

$\exists 0.c_0 \bullet P[0.c_0/0.c'] \wedge c \leq 0.c_0 \wedge c \leq c' \vee$

$\exists 1.c_0 \bullet P[1.c_0/1.c'] \wedge c \leq c' \wedge c \leq 1.c_0 \vee$

$\exists 0.c_0 \bullet P[0.c_0/0.c'] \wedge c \leq 0.c_0 \wedge c \leq c' \vee$

$\exists 1.c_0 \bullet P[1.c_0/1.c'] \wedge c \leq c' \wedge c \leq 1.c_0$

= {Propositional calculus}

$$\left(\begin{array}{l}\exists 0.c_0 \bullet P[0.c_0/0.c'] \wedge c \leq 0.c_0 \wedge c \leq c' \vee \\ \exists 1.c_0 \bullet P[1.c_0/1.c'] \wedge c \leq c' \wedge c \leq 1.c_0 \vee \\ \exists 0.c_0 \bullet P[0.c_0/0.c'] \wedge c \leq 0.c_0 \wedge c \leq c' \vee \\ \exists 1.c_0 \bullet P[1.c_0/1.c'] \wedge c \leq c' \wedge c \leq 1.c_0 \end{array}\right) \wedge c \leq c'$$

= {Inverse steps}

$(\mathbf{S1}[\mathbf{0}] \circ \mathbf{S1}[\mathbf{1}](P); M(c, 0.c, 1.c.c')) \wedge c \leq c'$

= {Definition 3.2.2}

$\mathbf{S1}(\mathbf{S1}[\mathbf{0}] \circ \mathbf{S1}[\mathbf{1}](P); M(c, 0.c, 1.c.c'))$ □

Law B.0.7.

$$\mathbf{S}[\mathbf{0}] \circ \mathbf{S}[\mathbf{1}](P_1 \vdash P_2); M = \mathbf{S}(\neg(\mathbf{S}[\mathbf{0}] \circ \mathbf{S}[\mathbf{1}](\neg P_1); M) \vdash \mathbf{S}[\mathbf{0}] \circ \mathbf{S}[\mathbf{1}](P_2); M)$$

Proof.

$\mathbf{S}[\mathbf{0}] \circ \mathbf{S}[\mathbf{1}](P_1 \vdash P_2); M$

= {Definition 2.3.1, propositional calculus}

$(\mathbf{S}[\mathbf{0}] \circ \mathbf{S}[\mathbf{1}](\neg ok) \vee \mathbf{S}[\mathbf{0}] \circ \mathbf{S}[\mathbf{1}](\neg P_1) \vee \mathbf{S}[\mathbf{0}] \circ \mathbf{S}[\mathbf{1}](ok' \wedge P_2)); M$

= {Disjunctivity of sequential composition}

$\mathbf{S}[\mathbf{0}] \circ \mathbf{S}[\mathbf{1}](\neg ok); M \vee \mathbf{S}[\mathbf{0}] \circ \mathbf{S}[\mathbf{1}](\neg P_1); M \vee \mathbf{S}[\mathbf{0}] \circ \mathbf{S}[\mathbf{1}](ok' \wedge P_2); M$

= {Laws B.0.5 and B.0.6, predicate calculus}

$\mathbf{S}(\neg ok) \vee \mathbf{S}(\mathbf{S}[\mathbf{0}] \circ \mathbf{S}[\mathbf{1}](\neg P_1); M) \vee \mathbf{S}(ok' \wedge \mathbf{S}[\mathbf{0}] \circ \mathbf{S}[\mathbf{1}](P_2); M)$

= {Propositional calculus}

$\mathbf{S}(ok \wedge \mathbf{S}(\mathbf{S}[\mathbf{0}] \circ \mathbf{S}[\mathbf{1}](\neg P_1); M) \Rightarrow ok' \wedge \mathbf{S}[\mathbf{0}] \circ \mathbf{S}[\mathbf{1}](P_2); M)$

= {Definition 2.3.1}

$\mathbf{S}(\neg(\mathbf{S}[\mathbf{0}] \circ \mathbf{S}[\mathbf{1}](\neg P_1); M) \vdash \mathbf{S}[\mathbf{0}] \circ \mathbf{S}[\mathbf{1}](P_2); M)$ □

Law 3.3.116 $P \parallel_{\mathrm{M}} Q = Q \parallel_{\mathrm{M}} P$

Proof.

$P \parallel_{\mathrm{M}} Q$

= {Definition 3.3.115, P and Q are $\mathbf{S}$-healthy}

$((\mathbf{S}(P); U0(m, c, x.out)) \parallel (\mathbf{S}(Q); U1(m, c, x.out))); M$

= {Laws 2.3.90, 2.3.115}

$((\mathbf{S}(Q); U1(m, c, x.out)) \parallel (\mathbf{S}(P); U0(m, c, x.out))); (0.m, 1.m := 1.m, 0.m); M$

= {Definition 2.2.3 and predicate calculus}

$((\mathbf{S}(Q); U0(m, c, x.out)) \parallel (\mathbf{S}(P); U1(m, c, x.out))); M$

= {Definition 3.3.115, P and Q are $\mathbf{S}$-healthy}

$Q \parallel_{\mathrm{M}} P$ □

Lemma B.0.8.

$$(\mathbf{S}[\mathbf{0}](P[\mathbf{0}]) \parallel \mathbf{S}[\mathbf{1}](R[\mathbf{1}])); M; U3(m) \parallel \mathbf{S}[\mathbf{2}](Q[\mathbf{2}]) = (\mathbf{S}[\mathbf{0}](P[\mathbf{0}]) \parallel \mathbf{S}[\mathbf{1}](R[\mathbf{1}]) \parallel \mathbf{S}[\mathbf{2}](Q[\mathbf{2}])); M; U3(m)$$

Proof.

$(\mathbf{S}[\mathbf{0}](P[\mathbf{0}]) \parallel \mathbf{S}[\mathbf{1}](R[\mathbf{1}])); M; U3(m) \parallel \mathbf{S}[\mathbf{2}](Q[\mathbf{2}])$

= {Law B.0.7 and separating simulation}

$\mathbf{S}[\mathbf{3}](\neg(\mathbf{S}[\mathbf{0}] \circ \mathbf{S}[\mathbf{1}](\neg P_1[\mathbf{0}] \vee \neg R_1[\mathbf{1}])); M[\mathbf{3}] \vdash \mathbf{S}[\mathbf{0}] \circ \mathbf{S}[\mathbf{1}](P_2[\mathbf{0}] \wedge R_2[\mathbf{1}]); M[\mathbf{3}]) \parallel \mathbf{S}[\mathbf{2}](Q[\mathbf{2}])$

= {Definition 3.3.107}

$\mathbf{S}[\mathbf{2}] \circ \mathbf{S}[\mathbf{3}](\neg(\mathbf{S}[\mathbf{0}] \circ \mathbf{S}[\mathbf{1}](\neg P_1[\mathbf{0}] \vee \neg R_1[\mathbf{1}]); M[\mathbf{3}]) \wedge Q_1[\mathbf{2}] \vdash (\mathbf{S}[\mathbf{0}] \circ \mathbf{S}[\mathbf{1}](P_2[\mathbf{0}] \wedge R_2[\mathbf{1}]); M; U3(m)) \wedge Q_2[\mathbf{2}])$

= {Definition 2.3.1, propositional calculus, theorem 3.2.13}

$\mathbf{S}[\mathbf{2}] \circ \mathbf{S}[\mathbf{3}](\neg(\mathbf{S}[\mathbf{0}] \circ \mathbf{S}[\mathbf{1}](\neg P_1[\mathbf{0}] \vee \neg R_1[\mathbf{1}]); M[\mathbf{3}]) \wedge \neg\mathbf{S}[\mathbf{3}](\neg Q_1[\mathbf{2}]) \vdash (\mathbf{S}[\mathbf{0}] \circ \mathbf{S}[\mathbf{1}](P_2[\mathbf{0}] \wedge R_2[\mathbf{1}]); M; U3(m)) \wedge Q_2[\mathbf{2}])$

= {Theorem 3.2.14, definition 2.2.3 and Q_2 does not mention the variables in $M; U3(m)$}

$\mathbf{S}[\mathbf{2}] \circ \mathbf{S}[\mathbf{3}](\neg(\mathbf{S}[\mathbf{0}] \circ \mathbf{S}[\mathbf{1}](\neg P_1[\mathbf{0}] \vee \neg R_1[\mathbf{1}]); M[\mathbf{3}]) \wedge \neg\mathbf{S}[\mathbf{3}](\neg Q_1[\mathbf{2}]) \vdash \mathbf{S}[\mathbf{0}] \circ \mathbf{S}[\mathbf{1}](P_2[\mathbf{0}] \wedge R_2[\mathbf{1}] \wedge Q_2[\mathbf{2}]); M; U3(m))$

= {Law B.0.5, propositional calculus}

$\mathbf{S}[\mathbf{2}] \circ \mathbf{S}[\mathbf{3}](\neg((\mathbf{S}[\mathbf{0}] \circ \mathbf{S}[\mathbf{1}](\neg P_1[\mathbf{0}] \vee \neg R_1[\mathbf{1}]); M[\mathbf{3}]) \vee (\mathbf{S}[\mathbf{0}] \circ \mathbf{S}[\mathbf{1}](\neg Q_1[\mathbf{2}]); M[\mathbf{3}])) \vdash \mathbf{S}[\mathbf{0}] \circ \mathbf{S}[\mathbf{1}](P_2[\mathbf{0}] \wedge R_2[\mathbf{1}] \wedge Q_2[\mathbf{2}]); M; U3(m))$

= {Disjunctivity of sequential composition}

$\mathbf{S}[\mathbf{2}] \circ \mathbf{S}[\mathbf{3}](\neg(\mathbf{S}[\mathbf{0}] \circ \mathbf{S}[\mathbf{1}](\neg P_1[\mathbf{0}] \vee \neg R_1[\mathbf{1}] \vee \neg Q_1[\mathbf{2}]); M; U3(m)) \vdash \mathbf{S}[\mathbf{0}] \circ \mathbf{S}[\mathbf{1}](P_2[\mathbf{0}] \wedge R_2[\mathbf{1}] \wedge Q_2[\mathbf{2}]); M; U3(m))$

= {Law B.0.7, propositional calculus}

$\mathbf{S}[\mathbf{0}] \circ \mathbf{S}[\mathbf{1}] \circ \mathbf{S}[\mathbf{2}](P_1[\mathbf{0}] \wedge R_1[\mathbf{1}] \wedge Q_1[\mathbf{2}] \vdash P_2[\mathbf{0}] \wedge R_2[\mathbf{1}] \wedge Q_2[\mathbf{2}]); M; U3(m)$

= {Definition 3.3.107}

$(\mathbf{S}[\mathbf{0}](P[\mathbf{0}]) \parallel \mathbf{S}[\mathbf{1}](R[\mathbf{1}]) \parallel \mathbf{S}[\mathbf{2}](Q[\mathbf{2}])); M; U3(m)$ □

Law 3.3.117 $P \parallel_{\mathrm{M}} (Q \parallel_{\mathrm{M}} R) = (P \parallel_{\mathrm{M}} Q) \parallel_{\mathrm{M}} R$

Proof.

Let $st =_{df} x, x.out$ in:

$(P \parallel_{\mathrm{M}} Q) \parallel_{\mathrm{M}} R$

= {Definition 2.3.104}

$((((P; U0) \parallel (Q; U1))_{+c,st}; \hat{M}; U0) \parallel (R; U2))_{+c,st}; M[2.x/1.x]$

= {Definition 3.3.107 and lemma B.0.8}

$(((P; U0) \parallel (Q; U1)) \parallel (R; U2))_{+c,st}; (\hat{M}; U0)_{+c,st}; M[2.x/1.x]$

= {Definition 2.2.3 and $M3$}

$((P; U0) \parallel (Q; U1) \parallel (R; U2))_{+c,st}; \hat{M}3$

= {Law 2.3.117}

$((P; U0) \parallel (Q; U1) \parallel (R; U2))_{+c,st}; (0.st, 1.st, 2.st := 1.st, 2.st, 0.st); \hat{M}3$

= {Definition 2.2.3 and predicate calculus}

$((P; U2) \parallel (Q; U0) \parallel (R; U1))_{+c,st}; \hat{M}3$

= {Law 3.3.108}

$((Q; U0) \parallel (R; U1) \parallel (P; U2))_{+c,st}; \hat{M}3$

= {Inverse argument}

$(Q \parallel_{\mathrm{M}} R) \parallel_{\mathrm{M}} P$

= {Law 3.3.116}

$P \parallel_{\mathrm{M}} (Q \parallel_{\mathrm{M}} R)$ □

Law 3.3.118 $(\text{Ⅱ} \parallel_{\mathrm{M}} P) \sqsubseteq P$

We begin by observing that, provided P does not mention $0.m$ we have:

$\mathbf{S1}[\mathbf{0}] \circ \mathbf{S1}[\mathbf{1}](P[\mathbf{1}]); M = \mathbf{S1}(P)$

We will show the proof for **S1**, the remaining healthiness conditions follow the same approach.

$\mathbf{S1}[\mathbf{0}] \circ \mathbf{S1}[\mathbf{1}](P[\mathbf{1}]); M(c, 0.c, 1.c, c')$

= {Definitions 3.2.1 and definition 2.3.114}

$P[1.c'/c'] \wedge c \leq 0.c' \wedge c \leq 1.c'; M(c, 0.c, 1.c, c')$

= {Case split}

$(c' = 1.c' = 0.c') \wedge P[1.c'/c'] \wedge c \leq 0.c' \wedge c \leq 1.c'; M(c, 0.c, 1.c, c') \vee$

$(c' = 0.c' \wedge c' \neq 1.c) \wedge P[1.c'/c'] \wedge c \leq 0.c \wedge c \leq 1.c'; M(c, 0.c, 1.c, c') \vee$

$(c' = 1.c \wedge c' \neq 0.c') \wedge P[1.c'/c'] \wedge c \leq 0.c' \wedge c \leq 1.c'; M(c, 0.c, 1.c, c') \vee$

$(1.c' \neq 0.c' \wedge c' \neq 1.c') \wedge P \wedge c \leq 0.c' \wedge c \leq 1.c'; M(c, 0.c, 1.c, c')$

= {Definition 2.2.3, law 2.3.118 and predicate calculus}

$P[1.c'/c'] \wedge c = 1.c' \wedge c = 0.c'; c' = c \wedge c' = 0.c \wedge c' = 1.c \vee$

$P[1.c'/c'] \wedge c \leq 1.c' \wedge c \leq 0.c'; c' = 1.c \vee$

$P[1.c'/c'] \wedge c \leq 1.c' \wedge c \leq 0.c'; c' = 0.c \vee$

$P[1.c'/c'] \wedge c \leq 1.c' \wedge c \leq 0.c'; c' = 0.c \sqcap 1.c$

= {Definition 2.2.9, disjunctivity of sequential composition, subsumption}

$P[1.c'/c'] \wedge c = 1.c' \wedge c \leq 0.c'; c' = c \wedge c' = 0.c \wedge c' = 1.c \vee$

$P[1.c'/c'] \wedge c \leq 1.c' \wedge c \leq 0.c'; c' = 1.c \vee$

$P[1.c'/c'] \wedge c \leq 1.c' \wedge c \leq 0.c'; c' = 0.c$

= {Definition 2.2.3}

$\exists 1.c_0, 0.c_0 \bullet P[1.c_0/c'] \wedge c \leq 1.c_0 \wedge c \leq 0.c_0 \wedge c' = c_0 \vee$

$\exists 1.c_0, 0.c_0 \bullet P[1.c_0/c'] \wedge c \leq 1.c_0 \wedge c \leq 0.c_0 \wedge c' = 1.c_0 \vee$

$\exists 1.c_0, 0.c_0 \bullet P[1.c_0/c'] \wedge c \leq 1.c_0 \wedge c \leq 0.c_0 \wedge c' = 0.c_0$

= {One-point rule}

$(P \wedge c \leq c') \vee (P \wedge c \leq c') \vee (\exists 1.c_0 \bullet P[1.c_0/c'] \wedge c \leq 1.c_0 \wedge c \leq c')$

$\sqsubseteq$ {Propositional calculus $(P \wedge (1.c_0 = c') \Rightarrow P)$, one-point rule}

$(P \wedge c \leq c') \vee (P \wedge c \leq c')$

= {Propositional calculus, definition 3.2.1}

$\mathbf{S1}(P)$

Law 3.3.119 $\bot \|_{\mathrm{M}} P = \bot$

Proof.

$\bot \|_{\mathrm{M}} P$

= {Definitions 3.3.2, 3.3.115 and 3.3.107}

$\mathbf{S}[\mathbf{0}] \circ \mathbf{S}[\mathbf{1}](\bot_D \parallel P[\mathbf{1}]); M$

= {Laws 2.3.93, theorem 2.3.3}

$\mathbf{S}[\mathbf{0}] \circ \mathbf{S}[\mathbf{1}](\mathbf{true}); M$

= {Law B.0.4 and defintion 3.3.2}

$\bot$ □

Law 3.3.120 $(P \lhd b \rhd Q) \|_{\mathrm{M}} R = ((P \|_{\mathrm{M}} R) \lhd b \rhd (Q \|_{\mathrm{M}} R))$

In order to keep the presentation compact, we will omit the variables affected by the separating simulations when this information is clear from the context.

Proof.

$(P \lhd b \rhd Q) \|_{\mathrm{M}} R$

= {Definition 3.3.115, P, Q and R are $\mathbf{S}$}

$(((P \lhd b \rhd Q); U0 \parallel (R; U1)); M$

= {Law 3.3.33, 2.3.94}

$(((P; U0) \parallel (R; U1)) \lhd b \rhd ((Q; U0) \parallel (R; U1))); M$

= {Law 3.3.33}

$(((P; U0) \parallel (R; U1)); M) \lhd b \rhd (((Q; U0) \parallel (R; U1)); M)$

= {Definition 3.3.115, P, Q and R are $\mathbf{S}$}

$(P \parallel_{\mathrm{M}} R) \lhd b \rhd (Q \parallel_{\mathrm{M}} R)$ □

Law 3.3.121 For any descending chain $P = \{P_n \mid n \in \mathbb{N}\}$ we have:

$$(\bigsqcap P) \parallel_{\mathrm{M}} R = \bigsqcap (P \parallel_{\mathrm{M}} R)$$

Proof.

$(\bigsqcap P) \parallel_{\mathrm{M}} R$

= {Definition 3.3.115}

$(((\bigsqcap P); U0) \parallel (R; U1)); M$

= {Laws 3.3.11 and 2.3.95}

$(((\bigsqcap (P; U0) \parallel (R; U1))); M$

= {Laws 3.3.11}

$\bigsqcap (((P; U0) \parallel (R; U1)); M)$

= {Definition 3.3.115}

$\bigsqcap (P \parallel_{\mathrm{M}} R)$ □

Law 3.3.122 $x \mathrel{:=}_{\mathrm{snc}} e_1 \parallel_{\mathrm{M}} x \mathrel{:=}_{\mathrm{snc}} e_2 \sqsubseteq x \mathrel{:=}_{\mathrm{snc}} \mathtt{SELECT}\,(e_1, e_2, x)$

Proof.

$x \mathrel{:=}_{\mathrm{snc}} e_1 \parallel_{\mathrm{M}} x \mathrel{:=}_{\mathrm{snc}} e_2$

= {Definitions 2.3.104, 3.3.13 and 2.3.101}

$((\mathbf{S}(\mathbf{true} \vdash x := e_1); \mathbf{var}\ 0.x := x; \mathbf{end}\ x) \parallel$

$\quad (\mathbf{S}(\mathbf{true} \vdash x := e_2); \mathbf{var}\ 1.x := x; \mathbf{end}\ x))_{+x}; M(x, 0.x, 1.x, x')$

= {Theorem 3.3.6, predicate and propositional calculus}

$(\mathbf{S}[\mathbf{0}](\mathbf{true} \vdash 0.x := e_1) \parallel \mathbf{S}[\mathbf{1}](\mathbf{true} \vdash 1.x := e_2)); M(x, 0.x, 1.x, x')$

= {Definition 2.3.89, propositional calculus and definition 3.3.13}

$\mathbf{S}[\mathbf{0}] \circ \mathbf{S}[\mathbf{1}](\mathbf{true} \vdash 0.x, 1.x \mathrel{:=}_{\mathrm{snc}} e_1, e_2); M(x, 0.x, 1.x, x')$

= {Definition 2.2.3, propositional calculus}

$$
\begin{array}{ll}
& \mathbf{S}(\mathbf{true} \vdash M(x, e_1, e_2, x')) \\
\sqsubseteq & \{\text{Theorem 2.3.122}\} \\
& x \overset{:=}{_{\mathrm{snc}}} \mathtt{SELECT}\,(0.x, 1.x, x)
\end{array}
$$

□

Theorem 3.3.123 Parallel by merge – **SH3** preservation

$$(P \parallel_{\mathrm{M}} Q); \Pi = (P \parallel_{\mathrm{M}} Q)$$

provided P and Q are **SH3**, **S**-healthy designs.

Proof.

$$
\begin{array}{ll}
& (P \parallel_{\mathrm{M}} Q); \Pi \\
= & \{\text{Definitions 3.3.115 and 2.3.101, } P \text{ and } Q \text{ are } \mathbf{S}\text{-healthy designs, predicate calculus}\} \\
& (\mathbf{S}[\mathbf{0}](P[\mathbf{0}]) \parallel_{\mathrm{M}} \mathbf{S}[\mathbf{1}](Q[\mathbf{1}])); M; \Pi \\
= & \{\text{Definition 3.3.107, law B.0.7}\} \\
& \mathbf{S}(\neg(\mathbf{S}[\mathbf{0}] \circ \mathbf{S}[\mathbf{1}](\neg(P_1[\mathbf{0}] \wedge Q_1[\mathbf{1}])); M) \vdash \mathbf{S}[\mathbf{0}] \circ \mathbf{S}[\mathbf{1}](P_2[\mathbf{0}] \wedge Q_2[\mathbf{1}]); M); \Pi \\
= & \{\text{Theorem 3.3.6, propositional calculus, theorem 3.2.9}\} \\
& \mathbf{S}(\neg(\mathbf{S}(\mathbf{S}[\mathbf{0}] \circ \mathbf{S}[\mathbf{1}](\neg(P_1[\mathbf{0}] \wedge Q_1[\mathbf{1}])); M); \mathbf{S}(\mathbf{true})) \vdash \mathbf{S}(\mathbf{S}[\mathbf{0}] \circ \mathbf{S}[\mathbf{1}](P_2[\mathbf{0}] \wedge Q_2[\mathbf{1}]); M); I\!I) \\
= & \{\text{Propositional calculus, disjunctivity of sequential composition, } P \text{ and } Q \text{ are } \mathbf{SH3}\} \\
& \mathbf{S}(\neg(\mathbf{S}(\mathbf{S}[\mathbf{0}] \circ \mathbf{S}[\mathbf{1}](\neg P_1[\mathbf{0}]); \mathbf{S}[\mathbf{0}] \circ \mathbf{S}[\mathbf{1}](\mathbf{true}); M \vee \\
& \qquad \mathbf{S}[\mathbf{0}] \circ \mathbf{S}[\mathbf{1}](\neg Q_1[\mathbf{1}]); \mathbf{S}[\mathbf{0}] \circ \mathbf{S}[\mathbf{1}](\mathbf{true}); M); \mathbf{S}(\mathbf{true})) \vdash \\
& \quad \mathbf{S}(\mathbf{S}[\mathbf{0}] \circ \mathbf{S}[\mathbf{1}](P_2[\mathbf{0}] \wedge Q_2[\mathbf{1}]); M); I\!I) \\
= & \{\text{Laws B.0.4 and 2.2.8}\} \\
& \mathbf{S}(\neg(\mathbf{S}(\mathbf{S}[\mathbf{0}] \circ \mathbf{S}[\mathbf{1}](\neg P_1[\mathbf{0}]); \mathbf{S}(\mathbf{true}) \vee \mathbf{S}[\mathbf{0}] \circ \mathbf{S}[\mathbf{1}](\neg Q_1[\mathbf{1}]); \mathbf{S}(\mathbf{true})); \mathbf{S}(\mathbf{true})) \vdash \\
& \quad \mathbf{S}(\mathbf{S}[\mathbf{0}] \circ \mathbf{S}[\mathbf{1}](P_2[\mathbf{0}] \wedge Q_2[\mathbf{1}]); M)) \\
= & \{\text{Disjunctivity of sequential composition, theorem 3.2.10}\} \\
& \mathbf{S}(\neg(\mathbf{S}(\mathbf{S}[\mathbf{0}] \circ \mathbf{S}[\mathbf{1}](\neg P_1[\mathbf{0}]) \vee \mathbf{S}[\mathbf{0}] \circ \mathbf{S}[\mathbf{1}](\neg Q_1[\mathbf{1}])); \mathbf{S}(\mathbf{true})) \vdash \\
& \quad \mathbf{S}(\mathbf{S}[\mathbf{0}] \circ \mathbf{S}[\mathbf{1}](P_2[\mathbf{0}] \wedge Q_2[\mathbf{1}]); M)) \\
= & \{\text{Law B.0.4}\} \\
& \mathbf{S}(\neg(\mathbf{S}(\mathbf{S}[\mathbf{0}] \circ \mathbf{S}[\mathbf{1}](\neg P_1[\mathbf{0}]) \vee \mathbf{S}[\mathbf{0}] \circ \mathbf{S}[\mathbf{1}](\neg Q_1[\mathbf{1}])); \mathbf{S}[\mathbf{0}] \circ \mathbf{S}[\mathbf{1}](\mathbf{true}); M) \vdash \\
& \quad \mathbf{S}(\mathbf{S}[\mathbf{0}] \circ \mathbf{S}[\mathbf{1}](P_2[\mathbf{0}] \wedge Q_2[\mathbf{1}]); M)) \\
= & \{\text{Inverse steps}\} \\
& \mathbf{S}(\neg(\mathbf{S}(\mathbf{S}[\mathbf{0}] \circ \mathbf{S}[\mathbf{1}](\neg(P_1[\mathbf{0}] \wedge Q_1[\mathbf{1}])); M)) \vdash \mathbf{S}(\mathbf{S}[\mathbf{0}] \circ \mathbf{S}[\mathbf{1}](P_2[\mathbf{0}] \wedge Q_2[\mathbf{1}]); M)) \\
= & \{\text{Definition 2.3.1, propositional calculus}\} \\
& \mathbf{S}(\mathbf{S}[\mathbf{0}] \circ \mathbf{S}[\mathbf{1}](\neg(P_1[\mathbf{0}] \wedge Q_1[\mathbf{1}])); M \vdash \mathbf{S}[\mathbf{0}] \circ \mathbf{S}[\mathbf{1}](P_2[\mathbf{0}] \wedge Q_2[\mathbf{1}]); M) \\
= & \{\text{Law B.0.7, definitions 3.3.107 and 2.3.101}\}
\end{array}
$$

$(\mathbf{S}[\mathbf{0}](P[\mathbf{0}]) \parallel_{\mathrm{M}} \mathbf{S}[\mathbf{1}](Q[\mathbf{1}])); M$

= {Definition 3.3.115, P and Q are $\mathbf{S}$-healthy}

$P \parallel_{\mathrm{M}} Q$ □

Theorem 3.3.125 The sync construct is **SH3**

$\mathbf{SH3}(\mathtt{sync})$

Proof.

Straightforward from definition 3.3.124 and theorem 3.3.15. □

Law 3.3.126 $\mathtt{sync}(\alpha)_{[x]} = \mathtt{sync}(\alpha \cup \{x\}) = \mathtt{sync}$

Proof.

Provided v is the set of variable names in α we have:

$\mathtt{sync}(v)_{[x]}$

= {Definitions 3.3.124 and 3.3.82}

$\mathbf{S}(\mathbf{true} \vdash c, v, v.out_c := c + 1, v.in_c, v \wedge$

$(x', x.out'_c, x.in' = (x.in_{c'-1}, x) \lhd c < c' \rhd (x, x.out_c), x.in) \wedge$

$\left\{x.out'_i = x.in_{i-1} \mid c < i < c'\right\} \wedge \{I\!I_{\{x.out_i\}} \mid i < c\})$

= {Definitions 2.3.9 and 2.2.6, propositional calculus}

$\mathbf{S}(\mathbf{true} \vdash c', v', v.out'_c = c + 1, v.in_c, v \wedge (x', x.out'_c = (x.in_{c'-1}, x) \wedge \{I\!I_{\{x.out_i\}} \mid i < c\}))$

= {Propositional calculus, definition 2.2.6}

$\mathbf{S}(\mathbf{true} \vdash c, v, v.out_c, x, x.out_c := c + 1, v.in_c, v, x.in_c, x)$

= {Definition 3.3.124}

$\mathtt{sync}(v, x)$

= {The extended alphabet α contains v as well as x}

sync □

Theorem 3.3.131 Final merge alternative formulation

$\hat{M} = R(x, 0.x, 1.x, x'); c := max(0.c, 1.c); \mathbf{end}\ 0.c, 1.c, 0.x, 1.x, 0.x.out, 1.x.out$

Proof.

Straightforward from law 2.3.11 and definition 3.3.130. □

Law 3.3.132 $(\dddot{0}.x, \dddot{1}.x := \dddot{1}.x, \dddot{0}.x); \hat{M} = \hat{M}$

Proof.

Straightforward from M's symmetry and the fact that max is commutative. □

Law 3.3.133 $(0.\dddot{x}, 1.\dddot{x}, 2.\dddot{x} := 1.\dddot{x}, 2.\dddot{x}, 0.\dddot{x}); \hat{M}3 = \hat{M}3$

Proof.

This result follows from is a simple but laborious case split on the possible combinations of c, $0.c$ and $1.c$. □

Law 3.3.135 $P \parallel_{\hat{M}} Q = Q \parallel_{\hat{M}} P$

Proof.

$$
\begin{aligned}
& P \parallel_{\hat{M}} Q \\
= & \{\text{Definition 3.3.134}\} \\
& ((P; U0(c, \ddot{\dddot{m}})) \parallel (Q; U1(c, \ddot{\dddot{m}})))_{+c,\ddot{\dddot{m}}}; \hat{M} \\
= & \{\text{Laws 3.3.108, and 3.3.132}\} \\
& ((Q; U1(c, \ddot{\dddot{m}})) \parallel (P; U0(c, \ddot{\dddot{m}})))_{+c,\ddot{\dddot{m}}}; (0.\dddot{m}, 1.\dddot{m} := 1.\dddot{m}, 0.\dddot{m}); \hat{M} \\
= & \{\text{Definition 2.3.101 and predicate calculus}\} \\
& (Q[\mathbf{1}] \parallel P[\mathbf{0}])_{+c,\ddot{\dddot{m}}}); (0.\dddot{m}, 1.\dddot{m} := 1.\dddot{m}, 0.\dddot{m}); \hat{M} \\
= & \{\text{Propositional calculus}\} \\
& (Q[\mathbf{0}] \parallel P[\mathbf{1}])_{+c,\ddot{\dddot{m}}}; \hat{M} \\
= & \{\text{Predicate calculus and definition 2.3.101}\} \\
& ((Q; U0(c, \ddot{\dddot{m}})) \parallel (P; U1(c, \ddot{\dddot{m}})))_{+c,\ddot{\dddot{m}}}; \hat{M} \\
= & \{\text{Definition 3.3.134}\} \\
& Q \parallel_{\hat{M}} P
\end{aligned}
$$

□

Law 3.3.136 $P \parallel_{\hat{M}} (Q \parallel_{\hat{M}} R) = (P \parallel_{\hat{M}} Q) \parallel_{\hat{M}} R$

For this proof we will need a similar set of results as provided in laws B.0.4 to B.0.7.

Lemma B.0.9.

$$\mathbf{S}[\mathbf{0}] \circ \mathbf{S}[\mathbf{1}](\boldsymbol{true}); \hat{M} = \mathbf{S}(\boldsymbol{true})$$

Proof.

The proof that $\hat{M}$ restores **S1** is similar to the proof of law B.0.4 but using *max* instead of M. The part of the proof dealing with **S2** is straightforward from the fact that $\hat{M}$ applies M to calculate the point-wise value for $x.out$ and that M satisfies the property we are trying to prove (law B.0.4). □

Lemma B.0.10. *Provided P does not mention the variables in the healthiness conditions we have:*

$$\mathbf{S}[\mathbf{0}] \circ \mathbf{S}[\mathbf{1}](P); \hat{M} = \mathbf{S}(P)$$

Proof.

Similar to the proof of lemma B.0.9. □

Lemma B.0.11. *Provided P mentions the variables in the merge predicate we have:*

$$\mathbf{S}[\mathbf{0}] \circ \mathbf{S}[\mathbf{1}](P); \hat{M} = \mathbf{S}(\mathbf{S}[\mathbf{0}] \circ \mathbf{S}[\mathbf{1}](P); \hat{M})$$

Proof.

Similar to the proof of lemma B.0.9. □

Lemma B.0.12.

$$\mathbf{S}[\mathbf{0}] \circ \mathbf{S}[\mathbf{1}](P_1 \vdash P_2); \hat{M} = \mathbf{S}(\neg(\mathbf{S}[\mathbf{0}] \circ \mathbf{S}[\mathbf{1}](\neg P_1); \hat{M}) \vdash \mathbf{S}[\mathbf{0}] \circ \mathbf{S}[\mathbf{1}](P_2); \hat{M})$$

Proof.

This proof follows the same outline of the proof of law B.0.7 but using lemma B.0.11 instead of B.0.6. □

With these results we can prove a similar result to the one expressed in lemma B.0.8 for the M predicate:

Lemma B.0.13.

$$(\mathbf{S}[\mathbf{0}](P[\mathbf{0}]) \parallel \mathbf{S}[\mathbf{1}](R[\mathbf{1}])); M; U3(m) \parallel \mathbf{S}[\mathbf{2}](Q[\mathbf{2}]) =$$
$$(\mathbf{S}[\mathbf{0}](P[\mathbf{0}]) \parallel \mathbf{S}[\mathbf{1}](R[\mathbf{1}]) \parallel \mathbf{S}[\mathbf{2}](Q[\mathbf{2}])); M; U3(m)$$

Proof.

This proof follows the same outline of the proof of lemma B.0.8 but using the corresponding results from this section. □

We can now prove that $\parallel_{\hat{M}}$ is associative:

Proof.

Let $st =_{df} x, x.out$ in:

$(P \parallel_{\hat{M}} Q) \parallel_{\hat{M}} R$
= {Definition 3.3.134}
$((((P; U0) \parallel (Q; U1))_{+c,st}; \hat{M}; U0) \parallel (R; U2))_{+c,st}; M[2.x/1.x]$
= {Definition 3.3.107 and lemma B.0.13}
$(((P; U0) \parallel (Q; U1)) \parallel (R; U2))_{+c,st}; (\hat{M}; U0)_{+c,st}; M[2.x/1.x]$
= {Definition 2.2.3 and $M3$}
$((P; U0) \parallel (Q; U1) \parallel (R; U2))_{+c,st}; \hat{M}3$
= {Law 2.3.117}
$((P; U0) \parallel (Q; U1) \parallel (R; U2))_{+c,st}; (0.st, 1.st, 2.st := 1.st, 2.st, 0.st); \hat{M}3$

= {Definition 2.2.3 and predicate calculus}

$((P; U2) \parallel (Q; U0) \parallel (R; U1))_{+c,st}; \hat{M}3$

= {Law 3.3.108}

$((Q; U0) \parallel (R; U1) \parallel (P; U2))_{+c,st}; \hat{M}3$

= {Inverse argument}

$(Q \parallel_{\hat{M}} R) \parallel_{\hat{M}} P$

= {Law 3.3.116}

$P \parallel_{\hat{M}} (Q \parallel_{\hat{M}} R)$ □

Law 3.3.137 $(\mathrm{II} \parallel_{\hat{M}} \mathrm{II}) = \mathrm{II}$

Proof.

$\mathrm{II} \parallel_{\hat{M}} \mathrm{II}$

= {Definition 3.3.134, predicate calculus}

$\mathbf{S}[\mathbf{0}] \circ \mathbf{S}[\mathbf{1}](I\!I_D[\mathbf{0}] \parallel I\!I_D[\mathbf{1}]); \hat{M}$

= {Law 3.3.110, definition 2.3.8}

$\mathbf{S}[\mathbf{0}] \circ \mathbf{S}[\mathbf{1}](\mathbf{true} \vdash I\!I_{\{0.c,1.c,0.m,1.m\}}); \hat{M}$

= {Definition 2.3.1, propositional calculus, disjunctivity of sequential composition}

$\left(\mathbf{S}[\mathbf{0}] \circ \mathbf{S}[\mathbf{1}](\neg ok); \hat{M}\right) \vee \left(\mathbf{S}[\mathbf{0}] \circ \mathbf{S}[\mathbf{1}](ok' \wedge I\!I_{\{0.c,1.c,0.m,1.m\}}); \hat{M}\right)$

= {Theorem 3.2.12, lemma B.0.10}

$\mathbf{S}(\neg ok) \vee \left(\mathbf{S}(ok') \wedge (\mathbf{S}[\mathbf{0}] \circ \mathbf{S}[\mathbf{1}](I\!I_{0.c,1.c,0.m,1.m}\{0.c, 1.c, 0.m, 1.m\}); \hat{M})\right)$

= {Theorem 3.2.8, definition 3.3.128 and predicate calculus}

$\mathbf{S}(\neg ok) \vee (\mathbf{S}(ok') \wedge I\!I)$

= {Theorems 3.2.12 and 3.2.13}

$\mathbf{S}(\neg ok \vee (ok' \wedge I\!I))$

= {Propositional calculus, definitions 2.3.1, 2.3.8 and 3.2.7}

II □

Law 3.3.138 $(\mathrm{II} \parallel_{\hat{M}} P) \sqsubseteq P$

Proof.

Similar to proof of law 2.3.108. □

Law 3.3.139 $\bot \parallel_{\hat{M}} P = \bot$

Proof.

$\bot \parallel_{\hat{M}} P$

= {Definitions 3.3.134, 2.3.101 and propositional calculus}

$(\mathbf{S}[\mathbf{0}](\mathbf{true}) \parallel \mathbf{S}[\mathbf{1}](P[\mathbf{1}])); \hat{M}$

= {Definition 3.3.107, law 2.3.93}

$\mathbf{S}[\mathbf{0}] \circ \mathbf{S}[\mathbf{1}](\mathbf{true}); \hat{M}$

= {Definition 3.3.128 predicate calculus}

$\mathbf{S}(\mathbf{true})$

= {Definition 3.3.2}

$\bot$ □

Law 3.3.140 $(P \lhd b \rhd Q) \parallel_{\hat{M}} R = ((P \parallel_{\hat{M}} R) \lhd b \rhd (Q \parallel_{\hat{M}} R))$

Proof.

$(P \lhd b \rhd Q) \parallel_{\hat{M}} R$

= {Definition 3.3.134}

$(((P \lhd b \rhd Q); U0(\ddot{m}, c)) \parallel (R; U1(\ddot{m}, c))); \hat{M}$

= {Law 2.3.24}

$((P[\mathbf{0}] \lhd b \rhd Q[\mathbf{0}]) \parallel R[\mathbf{1}]); \hat{M}$

= {Law 2.3.94}

$((P[\mathbf{0}] \parallel R[\mathbf{1}]) \lhd b \rhd (Q[\mathbf{0}] \parallel R[\mathbf{1}])); \hat{M}$

= {Law 2.3.24}

$((P[\mathbf{0}] \parallel R[\mathbf{1}]); \hat{M}) \lhd b \rhd (Q[\mathbf{0}] \parallel R[\mathbf{1}]); \hat{M})$

= {Definition 3.3.134}

$(P \parallel_{\hat{M}} R) \lhd b \rhd (Q \parallel_{\hat{M}} R)$ □

Law 3.3.141 For any descending chain $S = \{S_n \mid n \in \mathbb{N}\}$ we have: $(\bigsqcup S) \parallel_{\hat{M}} R = \bigsqcup_i (S_i \parallel_{\hat{M}} R)$

Proof.

$(\bigsqcup S) \parallel_{\hat{M}} R$

= {Definition 3.3.134}

$(((\bigsqcup S); U0) \parallel (R; U1)); \hat{M}$

= {Law 2.3.135}

$\left(\left(\left(\bigsqcup_i (S_i; U0)\right) \parallel (R; U1)\right); \hat{M}\right.$

= {Law 2.3.95}

$\left(\left(\bigsqcup_i ((S_i; U0) \parallel (R; U1))\right); \hat{M}\right.$

= {Law 2.3.135}

$\bigsqcup_i (((S_i; U0) \parallel (R; U1)); \hat{M})$

= {Definition 3.3.134}

$\bigsqcup_i (S_i \parallel_{\hat{M}} R)$ □

Law 3.3.142 Provided neither P nor Q perform any **sync** actions we have:

$$(P \parallel_M Q); \mathbf{sync}; (R \parallel_{\hat{M}} S) = (P; \mathbf{sync}; R) \parallel_{\hat{M}} (Q; \mathbf{sync}; S)$$

Proof.

This proof follows the same outline presented in [Hoare and He 1998, page 184]. □

Law 3.3.143 $b_{\mathsf{S}}^{\top}; (P \parallel_{\hat{M}} Q) = (b_{\mathsf{S}}^{\top}; P) \parallel_{\hat{M}} (b_{\mathsf{S}}^{\top}; Q)$

Proof.

$(b_{\mathsf{S}}^{\top}; P) \parallel_{\hat{M}} (b_{\mathsf{S}}^{\top}; Q)$

= {Definition 3.3.34, laws 3.3.33, 3.3.9, 3.3.7}

$(P \lhd b \rhd \bot) \parallel_{\hat{M}} (Q \lhd b \rhd \bot)$

= {3.3.140 (twice)}

$((P \parallel_{\hat{M}} Q) \lhd b \rhd (\bot \parallel_{\hat{M}} Q)) \lhd b \rhd ((P \parallel_{\hat{M}} \bot) \lhd b \rhd (\bot \parallel_{\hat{M}} \bot))$

= {3.3.26 (twice), 3.3.139}

$(P \parallel_{\hat{M}} Q) \lhd b \rhd \bot$

= {Laws 3.3.33, 3.3.9, 3.3.7 then definition 3.3.34}

$b_{\mathsf{S}}^{\top}; (P \parallel_{\hat{M}} Q)$ □

Law 3.3.144 $(P; (x = e)_{\mathsf{S}}^{\bot}) \parallel_{\hat{M}} Q_{[x]} = (P \parallel_{\hat{M}} Q_{[x]}); (x = e)_{\bot}$

Proof.

$(P \parallel_{\hat{M}} Q_{[x]}); (x = e)_{\mathsf{S}}^{\bot}$

= {Definition 3.3.134}

$((P; U0) \parallel (Q_{[x]}; U1))_{+m,c,\ddot{x}}; \hat{M}^{m,c,x}; (x = e)_{\mathsf{S}}^{\bot}$

= {Definitions 3.3.82 and 3.3.128, predicate calculus, law 3.3.16}

$((P; U0) \parallel (Q_{[x]}; U1))_{+m,c,\ddot{x}};$

$\quad \hat{M}^{m,c} \wedge x := 0.x \wedge x.out := 0.x.out; \mathbf{end}\ 0.x, 1.x, 0.x.out, 1.x.out; (x = e)_{\mathsf{S}}^{\bot}$

= {Laws 2.3.40, 2.3.48 and 2.3.11}

$((P; U0) \parallel (Q_{[x]}; U1))_{+m,c,\ddot{x}}; \hat{M}^{m,c} \wedge x := 0.x \wedge x.out := 0.x.out;$

$\quad (0.x = e)_{\mathsf{S}}^{\bot}; \mathbf{end}\ 0.x, 1.x, 0.x.out, 1.x.out$

= {Definition 3.3.107, lemma B.0.9, theorem 3.3.6 and predicate calculus}

$\mathbf{S}(\neg(\mathbf{S}[\mathbf{0}] \circ \mathbf{S}[\mathbf{1}](\neg(P_1[\mathbf{0}] \wedge Q_1[\mathbf{1}])); \hat{M}^{m,c} \wedge x := 0.x \wedge x.out := 0.x.out) \vdash$

$\mathbf{S}[\mathbf{0}] \circ \mathbf{S}[\mathbf{1}](\neg(P_2[\mathbf{0}] \wedge (Q_2[\mathbf{1}] \wedge E(x, c'))));$

$\hat{M}^{m,c} \wedge x := 0.x \wedge x.out := 0.x.out; (0.x = e)^{\top});$

end $0.x, 1.x, 0.x.out, 1.x.out$

= {Law 2.3.67, lemma B.0.9}

$\mathbf{S}[\mathbf{0}] \circ \mathbf{S}[\mathbf{1}](P_1[\mathbf{0}] \wedge Q_1[\mathbf{1}] \vdash (P_2[\mathbf{0}] \wedge Q_2[\mathbf{1}] \wedge E(x, c')); (0.x = e)^{\top});$

$\hat{M}^{m,c} \wedge x := 0.x \wedge x.out := 0.x.out$

end $0.x, 1.x, 0.x.out, 1.x.out$

= {Definition 2.2.3, propositional calculus}

$\mathbf{S}[\mathbf{0}] \circ \mathbf{S}[\mathbf{1}](P_1 \wedge Q_1 \vdash (P_2; (0.x = e)^{\top}) \wedge Q_2); \hat{M}^{m,c} \wedge x := 0.x \wedge x.out := 0.x.out$

end $0.x, 1.x, 0.x.out, 1.x.out$

= {Definition 3.3.107, predicate calculus and theorem 3.3.6}

$((P; U0; (0.x = e)^{\perp}_{\mathsf{S}}) \parallel (Q_{[x]}; U1))_{+m,c,\vec{x}};$

$\hat{M}^{m,c} \wedge x := 0.x \wedge x.out := 0.x.out;$ **end** $0.x, 1.x, 0.x.out, 1.x.out$

= {Laws 2.3.70 and 2.3.69}

$((P; (x = e)_{\perp}; U0) \parallel (Q_{[x]}; U1; \mathbf{var}\ \overset{\cdots}{1}.x))_{+m,c,\vec{x}};$

$\hat{M}^{m,c} \wedge x := 0.x \wedge x.out := 0.x.out;$ **end** $0.x, 1.x, 0.x.out, 1.x.out$

= {Inverse steps}

$(P; (x = e)^{\perp}_{\mathsf{S}}) \parallel_{\hat{M}} Q_{[x]}$ □

Law 3.3.145 $(x \overset{:=}{_{\mathsf{snc}}} e; \mathsf{sync}) = (x \overset{:=}{_{\mathsf{snc}}} e; \mathsf{sync}) \parallel_{\hat{M}} \mathsf{sync}$

Proof.

$(x \overset{:=}{_{\mathsf{snc}}} e; \mathsf{sync}) \parallel_{\hat{M}} \mathsf{sync}$

= {Definition 3.3.134}

$((x \overset{:=}{_{\mathsf{snc}}} e; \mathsf{sync}; U0(x, c)) \parallel (\mathsf{sync}; U1(x, c))); \hat{M}$

= {Definitions 3.3.13 and 3.3.124, law 3.3.16}

$((\mathbf{S}(x, x.out_c, c := x.in_c, e, c + 1); U0(x, c)) \parallel$

$(\mathbf{S}(x, x.out_c, c := x.in_c, x, c + 1); U1(x, c)))); \hat{M}$

= {Definition 3.3.128, laws 3.3.114, 3.3.16}

$$\mathbf{S}\left(\mathbf{true} \vdash \begin{array}{c} c := c + 1 \wedge M(x.in_{c+1-1}, x.in_c, x.in_c, x') \\ \wedge\ M(x, e, x, x.out'_c) \wedge \{\mathit{II}_{\{x.out_i\}} | i < c\} \end{array}\right); \mathbf{end}\ 0.c, 1.c, 0.x, 1.x, 0.x.out, 1.x.out$$

= {Predicate calculus, laws 2.3.119 and 2.3.118}

$\mathbf{S}(\mathbf{true} \vdash c' = c + 1 \wedge x' = x.in_c \wedge x.out'_c = e)$

= {Law 3.3.16}

$\mathbf{S}(\mathbf{true} \vdash x := e); \mathbf{S}(\mathbf{true} \vdash c, x, x.out_c := c + 1, x.in_c, x)$

= {Definitions 3.3.13 and 3.3.124}

$x \mathrel{:=}_{\mathsf{snc}} e; \mathsf{sync}$ □

Law 3.3.146 Provided x and y are different variables and that e_2 does not depend on x we have:

$$(x \mathrel{:=}_{\mathsf{snc}} e_1; \mathsf{sync}) \parallel_{\hat{M}} (y \mathrel{:=}_{\mathsf{snc}} e_2; \mathsf{sync}) = (x, y \mathrel{:=}_{\mathsf{snc}} e_1, e_2); \mathsf{sync}$$

Proof.

$(x \mathrel{:=}_{\mathsf{snc}} e_1; \mathsf{sync}) \parallel_{\hat{M}} (y \mathrel{:=}_{\mathsf{snc}} e_2; \mathsf{sync})$

= {Definitions 3.3.13 and 2.3.9}

$(x, y \mathrel{:=}_{\mathsf{snc}} e_1, y); \mathsf{sync} \parallel_{\hat{M}} (x, y \mathrel{:=}_{\mathsf{snc}} x, e_2); \mathsf{sync}$

= {Laws 3.3.10, 3.3.142 and 3.3.137; theorem 3.3.125 and law 3.3.10}

$((x, y \mathrel{:=}_{\mathsf{snc}} e_1, y) \parallel_{\mathrm{M}} (x, y \mathrel{:=}_{\mathsf{snc}} x, e_2)); \mathsf{sync}$

= {Definition 3.3.115, predicate calculus}

$\mathbf{S}(\mathbf{true} \vdash M(x, e_1, x, x') \wedge M(y, y, e_2, y')); \mathsf{sync}$

= {Law 2.3.119(twice), definitions 2.3.9 and 3.3.13}

$(x, y \mathrel{:=}_{\mathsf{snc}} e_1, e_2); \mathsf{sync}$ □

Law 3.3.147 Provided P and Q are **S**-healthy we have:

$$(P \parallel^{m}_{\hat{M}} Q)_{[x]} \sqsubseteq (P_{[x]} \parallel^{m,x}_{\hat{M}} Q_{[x]})$$

We will first show some auxiliary results that will help us structuring the proof of the result above in a more elegant way.

It is possible to split the effect of alphabet extension when combined with the variable being renamed by a separating simulation.

Lemma B.0.14. *Provided P is* **S***-healthy we have:*

$$P_{[x]}; U0(c, \ddot{x}) = P; U0(c); E_D(x, x', c, 0.c); U0(\ddot{x})$$

Proof.

$P_{[x]}; U0(c, \ddot{x})$

= {Definition 3.3.82 and 2.3.101}

$\mathbf{S}(P_1 \vdash P_2 \wedge (x', x.out'_c, x.in' = (x.in_{c'-1}, x) \lhd c < c' \rhd (x, x.out_c), x.in) \wedge$
$\left\{x.out'_i = x.in_{i-1} \mid c < i < c'\right\} \wedge \{I\!I_{\{x.out_i\}} \mid i < c\});$
$\mathbf{var}\ 0.c, \dddot{0.x} := c, \ddot{x}; \mathbf{end}\ c, \ddot{x}$

= {Definition 2.2.3, law 2.3.103 and propositional calculus}

$$\mathbf{S}[\mathbf{0}](P_1[\mathbf{0}] \vdash (P_2 \wedge (x', x.out'_c, x.in' = (x.in_{c'-1}, x) \lhd c < c' \rhd (x, x.out_c), x.in) \wedge \left\{x.out'_i = x.in_{i-1} \mid c < i < c'\right\} \wedge \{I\!I_{\{x.out_i\}} \mid i < c\}); U0(c); U0(\ddot{x}))$$

= {Propositional and predicate calculus}

$$\mathbf{S}[\mathbf{0}](P_1[\mathbf{0}] \vdash (P_2; U0(c); (x', x.out'_c, x.in' = (x.in_{0.c-1}, x) \lhd c < 0.c \rhd (x, x.out_c), x.in) \wedge \left\{x.out'_i = x.in_{i-1} \mid c < i < 0.c\right\} \wedge \{I\!I_{\{x.out_i\}} \mid i < c\}); U0(\ddot{x}))$$

= {Definition 2.2.3 and definition 3.3.84}

$$P; U0(c); E_D(x, x', c, 0.c); U0(\ddot{x})$$

□

Alphabet extension can be pushed half-way through a separating simulation.

Lemma B.0.15.

$$\textit{var}\ \ \dddot{0.x}; E_D(x, 0.x, c, 0.c); \textit{end}\ \ \ddot{x} = E_D(x, x', c, 0.c); U0(\ddot{x})$$

Proof.

$E_D(x, x', c, 0.c); U0(\ddot{x})$

= {Definition 2.3.101 and law 2.3.47}

$$\mathbf{var}\ \ \dddot{0.x}; E_D(x, x', c, 0.c)_{+\dddot{0.x}}; \dddot{0.x} := \dddot{x}; \mathbf{end}\ \ddot{x}$$

= {Definitions 3.3.84 and 2.2.3, propositional calculus}

$$\mathbf{var}\ \ \dddot{0.x}; (\mathbf{true} \vdash ((x', x.out'_c, x.in' = (x.in_{0.c-1}, x) \lhd c < 0.c \rhd (x, x.out_c), x.in) \wedge \left\{x.out'_i = x.in_{i-1} \mid c < i < 0.c\right\} \wedge \{I\!I_{\{x.out_i\}} \mid i < c \vee i \geq 0.c\} \wedge \dddot{0.x}' = \dddot{0.x}) ; (\dddot{0.x}', = \dddot{x})); \mathbf{end}\ \dddot{x}$$

= {Law 2.3.11}

$$\mathbf{var}\ \ \dddot{0.x}; \mathbf{S}(\mathbf{true} \vdash ((0.x', 0.x.out'_c, 0.x.in' = (x.in_{0.c-1}, x) \lhd c < 0.c \rhd (x, x.out_c), x.in) \wedge \left\{0.x.out'_i = x.in_{i-1} \mid c < i < 0.c\right\} \wedge \{I\!I_{\{0.x.out_i\}} \mid i < c \vee i \geq 0.c\}) ; (\dddot{0.x}', \dddot{x}' = \dddot{x}, \dddot{x})); \mathbf{end}\ \ddot{x}$$

= {Definition 3.3.84}

$$\mathbf{var}\ \ \dddot{0.x}; E_D(x, 0.x, c, 0.c); \mathbf{end}\ \ddot{x}$$

□

Applying the final merge predicate to two processes where the alphabet has been extended to include x is equivalent to applying the merge and extending the alphabet afterwards.

Lemma B.0.16.

$$\left(\begin{array}{l}(0.x', 0.x.out'_c, 0.x.in' = (x.in_{0.c-1}, x) \lhd c < 0.c \rhd (x, x.out_c), x.in) \wedge \\ \quad \left\{0.x.out'_i = x.in_{i-1} \mid c < i < 0.c\right\} \wedge \{\mathit{I\!I}_{\{0.x.out_i\}} \mid i < c \vee i \geq 0.c\} \ \wedge \\ (1.x', 1.x.out'_c, 1.x.in' = (x.in_{1.c-1}, x) \lhd c < 1.c \rhd (x, x.out_c), x.in) \wedge \\ \quad \left\{1.x.out'_i = x.in_{i-1} \mid c < i < 1.c\right\} \wedge \{\mathit{I\!I}_{\{1.x.out_i\}} \mid i < c \vee i \geq 1.c\}); \\ R(x, 0.x, 1.x, x')_{+c}; c := max(0.c, 1.c)\end{array}\right) = c := max(0.c, 1.c)_{[x]}$$

Proof.

By case analysis on $c = max(0.c, 1.c)$

Case $c = 0.c = 1.c$:

$$\begin{array}{l}((0.x', 0.x.out'_c, 0.x.in' = (x.in_{0.c-1}, x) \lhd c < 0.c \rhd (x, x.out_c), x.in) \wedge \\ \qquad \left\{0.x.out'_i = x.in_{i-1} \mid c < i < 0.c\right\} \wedge \{\mathit{I\!I}_{\{0.x.out_i\}} \mid i < c \vee i \geq 0.c\} \ \wedge \\ \qquad (1.x', 1.x.out'_c, 1.x.in' = (x.in_{1.c-1}, x) \lhd c < 1.c \rhd (x, x.out_c), x.in) \wedge \\ \qquad \left\{1.x.out'_i = x.in_{i-1} \mid c < i < 1.c\right\} \wedge \{\mathit{I\!I}_{\{1.x.out_i\}} \mid i < c \vee i \geq 1.c\}); \\ (0.x.out_{0.c}, 1.x.out_{1.c} := 0.x, 1.x); \\ \left(\begin{array}{l}M(x_{0.c,1.c}, (0.x \lhd 0.c \geq 1.c \rhd x_{0.c,1.c}), (1.x \lhd 1.c \geq 0.c \rhd x_{0.c,1.c}), x') \wedge \\ \{M(x.in_{i-1}, (0.out_i \lhd 0.c \geq i \rhd in_{i-1}), \\ \qquad (1.out_i \lhd 1.c \geq i \rhd in_{i-1}), x.out'_i) \mid c < i < max(0.c, 1.c)\} \wedge \\ M(x, 0.x.out_c, 1.x.out_c, x.out'_c) \lhd c < max(0.c, 1.c) \rhd (x.out_c := x.out_c) \wedge \\ \{\mathit{I\!I}_{x.out_i} \mid i < c\})\end{array}\right); \\ c := max(0.c, 1.c) \\ = \{c = 0.c = 1.c, \text{propositional calculus}\} \\ (0.x', 0.x.out', 0.x.in' = x, x.out, x.in) \wedge (1.x', 1.x.out', 1.x.in' = x, x.out, x.in); \\ (0.x.out_{0.c}, 1.x.out_{1.c} := 0.x, 1.x); \\ \left(\begin{array}{l}M(x_{0.c,1.c}, (0.x \lhd 0.c \geq 1.c \rhd x_{0.c,1.c}), (1.x \lhd 1.c \geq 0.c \rhd x_{0.c,1.c}), x') \wedge \\ \{M(x.in_{i-1}, (0.out_i \lhd 0.c \geq i \rhd in_{i-1}), \\ \qquad (1.out_i \lhd 1.c \geq i \rhd in_{i-1}), x.out'_i) \mid c < i < max(0.c, 1.c)\} \wedge \\ M(x, 0.x.out_c, 1.x.out_c, x.out'_c) \lhd c < max(0.c, 1.c) \rhd (x.out_c := x.out_c) \wedge \\ \{\mathit{I\!I}_{x.out_i} \mid i < c\}\end{array}\right); \\ c := max(0.c, 1.c) \\ = \{\text{Law 2.3.11 and propositional calculus}\} \\ (M(x, x, x, x') \wedge x.out := x.out); c := max(0.c, 1.c) \\ = \{\text{Laws 2.3.119 and 2.3.11, propositional calculus}\}\end{array}$$

$$
\begin{array}{l}
\quad c := max(0.c, 1.c) \\
= \{c = 0.c = 1.c, \text{propositional calculus}\} \\
\quad c' = max(0.c, 1.c) \wedge \\
\qquad (x', x.out'_c, x.in' = (x.in_{c'-1}, x) \lhd c < c' \rhd (x, x.out_c), x.in) \wedge \\
\qquad \left\{x.out'_i = x.in_{i-1} \mid c < i < c'\right\} \wedge \{\mathbb{I}_{\{x.out_i\}} \mid i < c\} \\
= \{\text{Definition 3.3.82}\} \\
\quad c := max(0.c, 1.c)_{[x]}
\end{array}
$$

For the remaining case ($c < max(0.c = 1.c)$) we will further split on the relationship between $0.c$ and $1.c$. In particular, the proof explores the cases where $1.c$ is greater than c and all the possible values of $0.c$ (i.e., $0.c = 1.c$, $c < 0.c < 1.c$ and $c = 0.c \wedge 0.c < 1.c$). The remaining sub-cases are symmetric and we do not show them here.

Sub-case $0.c = 1.c \wedge c < 0.c$:

$$
\begin{array}{l}
((0.x', 0.x.out'_c, 0.x.in' = (x.in_{0.c-1}, x) \lhd c < 0.c \rhd (x, x.out_c), x.in) \wedge \\
\qquad \left\{0.x.out'_i = x.in_{i-1} \mid c < i < 0.c\right\} \wedge \{\mathbb{I}_{\{0.x.out_i\}} \mid i < c \vee i \geq 0.c\} \ \wedge \\
\qquad (1.x', 1.x.out'_c, 1.x.in' = (x.in_{1.c-1}, x) \lhd c < 1.c \rhd (x, x.out_c), x.in) \wedge \\
\qquad \left\{1.x.out'_i = x.in_{i-1} \mid c < i < 1.c\right\} \wedge \{\mathbb{I}_{\{1.x.out_i\}} \mid i < c \vee i \geq 1.c\}); \\
(0.x.out_{0.c}, 1.x.out_{1.c} := 0.x, 1.x); \\
\left(\begin{array}{l}
M(x_{0.c,1.c}, (0.x \lhd 0.c \geq 1.c \rhd x_{0.c,1.c}), (1.x \lhd 1.c \geq 0.c \rhd x_{0.c,1.c}), x') \wedge \\
\{M(x.in_{i-1}, (0.out_i \lhd 0.c \geq i \rhd in_{i-1}), \\
\quad (1.out_i \lhd 1.c \geq i \rhd in_{i-1}), x.out'_i) \mid c < i < max(0.c, 1.c)\} \wedge \\
M(x, 0.x.out_c, 1.x.out_c, x.out'_c) \lhd c < max(0.c, 1.c) \rhd (x.out_c := x.out_c) \wedge \\
\{\mathbb{I}_{x.out_i} \mid i < c\}
\end{array}\right); \\
\quad c := max(0.c, 1.c) \\
= \{0.c = 1.c \wedge c < 0.c, \text{propositional and predicate calculus}\} \\
\quad ((0.x', 0.x.out'_c, 0.x.in' = x.in_{0.c-1}, x, x.in) \wedge \\
\qquad \left\{0.x.out'_i = x.in_{i-1} \mid c < i < 0.c\right\} \wedge \{\mathbb{I}_{\{0.x.out_i\}} \mid i < c \vee i \geq 0.c\} \ \wedge \\
\qquad (1.x', 1.x.out'_c, 1.x.in' = x.in_{1.c-1}, x, x.in) \wedge \\
\qquad \left\{1.x.out'_i = x.in_{i-1} \mid c < i < 1.c\right\} \wedge \{\mathbb{I}_{\{1.x.out_i\}} \mid i < c \vee i \geq 1.c\}); \\
\left(\begin{array}{l}
M(x.in_{0.c-1}, 0.x, 1.x, x') \wedge \{M(in_{i-1}, 0.out_i, 1.out_i, x.out'_i) \mid c < i < 0.c\} \wedge \\
M(x, 0.x.out_c, 1.x.out_c, x.out'_c) \wedge \{\mathbb{I}_{x.out_i} \mid i < c\}
\end{array}\right); \\
\quad c := max(0.c, 1.c) \\
= \{\text{Law 2.3.11 and propositional calculus}\} \\
\quad (M(x.in_{0.c-1}, x.in_{0.c-1}, x.in_{0.c-1}, x') \wedge \{M(in_{i-1}, in_{i-1}, in_{i-1}, x.out'_i) \mid c < i < 0.c\} \wedge
\end{array}
$$

$$M(x, x, x, x.out'_c) \wedge \{I\!I_{x.out_i} \mid i < c\}); c := max(0.c, 1.c)$$

$$= \{\text{Law 2.3.119, predicate calculus and law 2.3.11}\}$$

$$((c', x', x.out'_c = 0.c, x.in_{0.c-1}, x) \wedge \{M(in_{i-1}, in_{i-1}, in_{i-1}, x.out'_i) \mid c < i < 0.c\})$$

$$= \{c' = 0.c\}$$

$$(c', x', x.out'_c = 0.c, x.in_{c'-1}, x) \wedge \{M(in_{i-1}, in_{i-1}, in_{i-1}, x.out'_i) \mid c < i < c'\}$$

$$= \{0.c = 1.c \wedge c < 0.c, \text{propositional calculus}\}$$

$$\begin{aligned}(c' = max(0.c, 1.c) \wedge \\ (x', x.out'_c, x.in' = (x.in_{c'-1}, x) \lhd c < c' \rhd (x, x.out_c), x.in) \wedge \\ \left\{x.out'_i = x.in_{i-1} \mid c < i < c'\right\} \wedge \{I\!I_{\{x.out_i\}} \mid i < c\})\end{aligned}$$

$$= \{\text{Definition 3.3.82}\}$$

$$c := max(0.c, 1.c)_{[x]}$$

Sub-case $c < 0.c < 1.c$:

$$\begin{aligned}&((0.x', 0.x.out'_c, 0.x.in' = (x.in_{0.c-1}, x) \lhd c < 0.c \rhd (x, x.out_c), x.in) \wedge \\ &\qquad \left\{0.x.out'_i = x.in_{i-1} \mid c < i < 0.c\right\} \wedge \{I\!I_{\{0.x.out_i\}} \mid i < c \vee i \geq 0.c\} \wedge \\ &\qquad (1.x', 1.x.out'_c, 1.x.in' = (x.in_{1.c-1}, x) \lhd c < 1.c \rhd (x, x.out_c), x.in) \wedge \\ &\qquad \left\{1.x.out'_i = x.in_{i-1} \mid c < i < 1.c\right\} \wedge \{I\!I_{\{1.x.out_i\}} \mid i < c \vee i \geq 1.c\}); \\ &(0.x.out_{0.c}, 1.x.out_{1.c} := 0.x, 1.x); \\ &\left(\begin{array}{l} M(x_{0.c,1.c}, (0.x \lhd 0.c \geq 1.c \rhd x_{0.c,1.c}), (1.x \lhd 1.c \geq 0.c \rhd x_{0.c,1.c}), x') \wedge \\ \{M(x.in_{i-1}, (0.out_i \lhd 0.c \geq i \rhd in_{i-1}), \\ \quad (1.out_i \lhd 1.c \geq i \rhd in_{i-1}), x.out'_i) \mid c < i < max(0.c, 1.c)\} \wedge \\ M(x, 0.x.out_c, 1.x.out_c, x.out'_c) \lhd c < max(0.c, 1.c) \rhd (x.out_c := x.out_c) \wedge \\ \{I\!I_{x.out_i} \mid i < c\} \end{array}\right); c := max(0.c, 1.c)\end{aligned}$$

$$= \{c < 0.c < 1.c, \text{propositional and predicate calculus}\}$$

$$\begin{aligned}&((0.x', 0.x.out'_c, 0.x.in' = x.in_{0.c-1}, x, x.in) \wedge \\ &\qquad \left\{0.x.out'_i = x.in_{i-1} \mid c < i < 0.c\right\} \wedge \{I\!I_{\{0.x.out_i\}} \mid i < c \vee i \geq 0.c\} \wedge \\ &\qquad (1.x', 1.x.out'_c, 1.x.in' = x.in_{1.c-1}, x, x.in) \wedge \\ &\qquad \left\{1.x.out'_i = x.in_{i-1} \mid c < i < 1.c\right\} \wedge \{I\!I_{\{1.x.out_i\}} \mid i < c \vee i \geq 1.c\}); \\ &\left(\begin{array}{l} M(x.in_{1.c-1}, x.in_{1.c-1}, 1.x, x') \wedge \\ \{M(x.in_{i-1}, 0.x.out_i, 1.x.out_i, x.out'_i) \mid c < i < 0.c\} \wedge \\ M(x.in_{0.c-1}, x.in_{0.c-1}, 1.x.out_{0.c}, x.out'_i) \wedge \\ \{M(x.in_{i-1}, x.in_{i-1}, 1.x.out_i, x.out'_i) \mid 0.c < i < 1.c\} \wedge \\ M(x, 0.x.out_c, 1.x.out_c, x.out'_c) \wedge \{I\!I_{x.out_i} \mid i < c\}) \end{array}\right); c := max(0.c, 1.c)\end{aligned}$$

$$= \{\text{Predicate calculus, law 2.3.11}\}$$

$$\left(\begin{array}{l} M(x.in_{1.c-1}, x.in_{1.c-1}, x.in_{1.c-1}, x')\wedge \\ \{M(x.in_{i-1}, x.in_{i-1}, x.in_{i-1}, x.out'_i) \mid c < i < 0.c\}\wedge \\ M(x.in_{0.c-1}, x.in_{0.c-1}, x.out_{0.c-1}, x.out'_i)\wedge \\ \{M(x.in_{i-1}, x.in_{i-1}, x.in_{i-1}, x.out'_i) \mid 0.c < i < 1.c\}\wedge \\ M(x, x, x, x.out'_c) \wedge \{I\!I_{x.out_i} \mid i < c\}) \end{array}\right); c := max(0.c, 1.c)$$

= {Law 2.3.119, predicate calculus, law 2.3.11}

$$(c', x', x.out'_c = 1.c, x.in_{1.c-1}, x) \wedge \left\{x.out'_i = x.in_{i-1} \mid c < i < 1.c\right\}$$

= {$c' = 1.c$ and $c < 0.c < 1.c$, propositional calculus}

$$c' = max(0.c, 1.c) \wedge (x', x.out'_c, x.in' = (x.in_{c'-1}, x) \lhd c < c' \rhd (x, x.out_c), x.in) \wedge \left\{x.out'_i = x.in_{i-1} \mid c < i < c'\right\} \wedge \{I\!I_{\{x.out_i\}} \mid i < c\}$$

= {Definition 3.3.82}

$$c := max(0.c, 1.c)_{[x]}$$

Sub-case $c = 0.c \wedge 0.c < 1.c$:

$$\begin{array}{l} ((0.x', 0.x.out'_c, 0.x.in' = (x.in_{0.c-1}, x) \lhd c < 0.c \rhd (x, x.out_c), x.in) \wedge \\ \qquad \left\{0.x.out'_i = x.in_{i-1} \mid c < i < 0.c\right\} \wedge \{I\!I_{\{0.x.out_i\}} \mid i < c \vee i \geq 0.c\} \ \wedge \\ \qquad (1.x', 1.x.out'_c, 1.x.in' = (x.in_{1.c-1}, x) \lhd c < 1.c \rhd (x, x.out_c), x.in) \wedge \\ \qquad \left\{1.x.out'_i = x.in_{i-1} \mid c < i < 1.c\right\} \wedge \{I\!I_{\{1.x.out_i\}} \mid i < c \vee i \geq 1.c\}); \\ (0.x.out_{0.c}, 1.x.out_{1.c} := 0.x, 1.x); \\ \left(\begin{array}{l} M(x_{0.c,1.c}, (0.x \lhd 0.c \geq 1.c \rhd x_{0.c,1.c}), (1.x \lhd 1.c \geq 0.c \rhd x_{0.c,1.c}), x')\wedge \\ \{M(x.in_{i-1}, (0.out_i \lhd 0.c \geq i \rhd in_{i-1}), \\ \quad (1.out_i \lhd 1.c \geq i \rhd in_{i-1}), x.out'_i) \mid c < i < max(0.c, 1.c)\}\wedge \\ M(x, 0.x.out_c, 1.x.out_c, x.out'_c) \lhd c < max(0.c, 1.c) \rhd (x.out_c := x.out_c)\wedge \\ \{I\!I_{x.out_i} \mid i < c\} \end{array}\right); \\ c := max(0.c, 1.c) \end{array}$$

= {$c = 0.c \wedge 0.c < 1.c$, propositional and predicate calculus}

$$\begin{array}{l} ((0.x', 0.x.out'_c, 0.x.in' = x, x.out_c, x.in) \wedge \\ \qquad (1.x', 1.x.out'_c, 1.x.in' = x.in_{1.c-1}, x, x.in) \wedge \\ \qquad \left\{1.x.out'_i = x.in_{i-1} \mid c < i < 1.c\right\} \wedge \{I\!I_{\{1.x.out_i\}} \mid i < c \vee i \geq 1.c\}); \\ (0.x.out_{0.c}, 1.x.out_{1.c} := 0.x, 1.x); \\ \left(\begin{array}{l} M(x.in_{1.c-1}, x.in_{1.c-1}, 1.x, x')\wedge \\ \{M(x.in_{i-1}, x.in_{i-1}, 1.x.out_i, x.out'_i) \mid c < i < 1.c\}\wedge \\ M(x, 0.x.out_c, 1.x.out_c, x.out'_c) \wedge \{I\!I_{x.out_i} \mid i < c\} \end{array}\right); c := max(0.c, 1.c) \end{array}$$

= {Predicate calculus, law 2.3.11}

$$\left(\begin{array}{l} M(x.in_{1.c-1}, x.in_{1.c-1}, x.in_{1.c-1}, x') \wedge \\ \{M(x.in_{i-1}, x.in_{i-1}, x.in_{i-1}, x.out_i') \mid c < i < 1.c\} \wedge \\ M(x, x, 1.x.out_c, x.out_c') \wedge \{\mathit{I\!I}_{x.out_i} \mid i < c\}) \end{array}\right); c := max(0.c, 1.c)$$

= {Law 2.3.119, predicate calculus, law 2.3.11}

$(c', x', x.out_c' = 1.c, x.in_{1.c-1}, x) \wedge \left\{x.out_i' = x.in_{i-1} \mid c < i < 1.c\right\}$

= $\{c' = 1.c\}$

$(c', x', x.out_c' = 1.c, x.in_{c'-1}, x) \wedge \left\{x.out_i' = x.in_{i-1} \mid c < i < c'\right\}$

= {$c < 0.c < 1.c$, propositional calculus}

$(c' = max(0.c, 1.c) \wedge$

$(x', x.out_c', x.in' = (x.in_{c'-1}, x) \lhd c < c' \rhd (x, x.out_c), x.in) \wedge$

$\left\{x.out_i' = x.in_{i-1} \mid c < i < c'\right\} \wedge \{\mathit{I\!I}_{\{x.out_i\}} \mid i < c\})$

= {Definition 3.3.82}

$c := max(0.c, 1.c)_{[x]}$ □

Lemma B.0.17.

$\mathbf{S}[\mathbf{0}] \circ \mathbf{S}[\mathbf{1}](\mathit{I\!I}_D); c := max(0.c, 1.c)_{[x]} =$

$(\mathbf{S}[\mathbf{0}](E_D(x, 0.x, c, 0.c)) \parallel \mathbf{S}[\mathbf{1}](E_D(x, 1.x, c, 1.c))); R(x, 0.x, 1.x, x')_{+c}; c := max(0.c, 1.c)$

Proof.

$(\mathbf{S}[\mathbf{0}](E_D(x, 0.x, c, 0.c)) \parallel \mathbf{S}[\mathbf{1}](E_D(x, 1.x, c, 1.c))); R(x, 0.x, 1.x, x')_{+c}; c := max(0.c, 1.c)$

= {Definitions 3.3.84 and 3.3.107}

$\mathbf{S}[\mathbf{0}] \circ \mathbf{S}[\mathbf{1}]((\mathbf{true} \vdash (0.x', 0.x.out_c', 0.x.in' = (x.in_{0.c-1}, x) \lhd c < 0.c \rhd (x, x.out_c), x.in) \wedge$

$\left\{0.x.out_i' = x.in_{i-1} \mid c < i < 0.c\right\} \wedge \{\mathit{I\!I}_{\{0.x.out_i\}} \mid i < c \vee i \geq 0.c\} \wedge$

$(1.x', 1.x.out_c', 1.x.in' = (x.in_{1.c-1}, x) \lhd c < 1.c \rhd (x, x.out_c), x.in) \wedge$

$\left\{1.x.out_i' = x.in_{i-1} \mid c < i < 1.c\right\} \wedge \{\mathit{I\!I}_{\{1.x.out_i\}} \mid i < c \vee i \geq 1.c\}));$

$R(x, 0.x, 1.x, x')_{+c}; c := max(0.c, 1.c)$

= {Definition 2.2.3, predicate calculus}

$\mathbf{S}[\mathbf{0}] \circ \mathbf{S}[\mathbf{1}]((\mathbf{true} \vdash (0.x', 0.x.out_c', 0.x.in' = (x.in_{0.c-1}, x) \lhd c < 0.c \rhd (x, x.out_c), x.in) \wedge$

$\left\{0.x.out_i' = x.in_{i-1} \mid c < i < 0.c\right\} \wedge \{\mathit{I\!I}_{\{0.x.out_i\}} \mid i < c \vee i \geq 0.c\} \wedge$

$(1.x', 1.x.out_c', 1.x.in' = (x.in_{1.c-1}, x) \lhd c < 1.c \rhd (x, x.out_c), x.in) \wedge$

$\left\{1.x.out_i' = x.in_{i-1} \mid c < i < 1.c\right\} \wedge \{\mathit{I\!I}_{\{1.x.out_i\}} \mid i < c \vee i \geq 1.c\});$

$R(x, 0.x, 1.x, x')_{+c}; c := max(0.c, 1.c))$

= {Lemma B.0.16}

$\mathbf{S}[\mathbf{0}] \circ \mathbf{S}[\mathbf{1}](\mathbf{true} \vdash c := max(0.c, 1.c)_{[x]})$

= {Definition 2.2.3, predicate calculus}

$\mathbf{S}[\mathbf{0}] \circ \mathbf{S}[\mathbf{1}](I\!I_D); c := max(0.c, 1.c)_{[x]}$ □

Lemma B.0.18.

$$\hat{M}^{m,c}_{[x]}; \textbf{\textit{end}} \;\; \dddot{0}.x, \dddot{1}.x \sqsubseteq \mathbf{S}[\mathbf{0}] \circ \mathbf{S}[\mathbf{1}](E_D(x, 0.x, c, 0.c) \parallel E_D(x, 1.x, c, 1.c)); \hat{M}^{m,c,x}$$

Proof.

$\hat{M}^{m,c}_{[x]}; \mathbf{end} \;\; \dddot{0}.x, \dddot{1}.x$

⊑ {Theorem 3.3.131 and law 3.3.90}

$(R(m, 0.m, 1.m, m')_{+c})_{[x]}; (c := max(0.c, 1.c))_{[x]}; \mathbf{end}\; 0.m, 1.m, 0.c.1, c, \dddot{0}.x, \dddot{1}.x$

= {Law 3.3.87 and lemma B.0.17}

$R(m, 0.m, 1.m, m')_{+c,x,x.in,x.out}; \mathbf{S}[\mathbf{0}] \circ \mathbf{S}[\mathbf{1}](E_D(x, 0.x, c, 0.c) \parallel E_D(x, 1.x, c, 1.c));$

$R(x, 0.x, 1.x, x')_{+c}; c := max(0.c, 1.c); \mathbf{end}\; 0.m, 1.m, 0.c.1, c, \dddot{0}.x, \dddot{1}.x$

= {The variables in $\mathbf{S}[\mathbf{0}] \circ \mathbf{S}[\mathbf{1}](E_D(x, 0.x, c, 0.c) \parallel E_D(x, 1.x, c, 1.c))$
are not mentioned in $R(m, 0.m, 1.m, m')$, law 2.3.15}

$\mathbf{S}[\mathbf{0}] \circ \mathbf{S}[\mathbf{1}](E_D(x, 0.x, c, 0.c) \parallel E_D(x, 1.x, c, 1.c)); R(m, 0.m, 1.m, m')_{+c,x,x.in,x.out};$

$R(x, 0.x, 1.x, x')_{+c}; c := max(0.c, 1.c); \mathbf{end}\; 0.m, 1.m, 0.c.1, c, \dddot{0}.x, \dddot{1}.x$

= {Definition 3.3.128}

$\mathbf{S}[\mathbf{0}] \circ \mathbf{S}[\mathbf{1}](E_D(x, 0.x, c, 0.c) \parallel E_D(x, 1.x, c, 1.c)); \hat{M}^{m,c,x}$ □

We can now prove law 3.3.147 Provided P and Q are $\mathbf{S}$-healthy we have:

$$(P \parallel^m_{\hat{M}} Q)_{[x]} \sqsubseteq (P_{[x]} \parallel^{m,x}_{\hat{M}} Q_{[x]})$$

Proof.

$(P \parallel^m_{\hat{M}} Q)_{[x]}$

⊑ {Definition 3.3.134, law 3.3.90}

$(((P; U0(c, \ddot{m})) \parallel (Q; U1(c, \ddot{m})))_{+c,\ddot{m}})_{[x]}; \hat{M}_{[x]}$

= {Laws 3.3.87, 2.3.37 and 2.3.48}

$((P; U0(c, \ddot{m})) \parallel (Q; U1(c, \ddot{m})))_{+c,\ddot{m},\ddot{x}}; \mathbf{var}\;\; \dddot{0}.x; \mathbf{var}\;\; \dddot{1}.x; \hat{M}_{[x]}; \mathbf{end}\;\; \dddot{0}.x, \dddot{1}.x$

⊑ {Lemma B.0.18}

$(((P; U0(c, \ddot{m})) \parallel (Q; U1(c, \ddot{m})))_{+c,\ddot{m},\ddot{x}}; \mathbf{var}\;\; \dddot{0}.x; \mathbf{var}\;\; \dddot{1}.x;$

$\mathbf{S}[\mathbf{0}] \circ \mathbf{S}[\mathbf{1}](E_D(x, 0.x, c, 0.c) \parallel E_D(x, 1.x, c, 1.c)); \hat{M}^{m,c,x}$

= {Predicate calculus}

$(((P; U0(c, \ddot{m})) \parallel (Q; U1(c, \ddot{m})))_{+c,\ddot{m},\ddot{x}};$

$$
\begin{array}{ll}
 & ((\mathbf{var}\ \dddot{0}.x; E_D(x, 0.x, c, 0.c)) \parallel (\mathbf{var}\ \dddot{1}.x; E_D(x, 1.x, c, 1.c))); \hat{M}^{m,c,x} \\
= & \{\text{Predicate calculus}\} \\
 & ((P; U0(c, \ddot{m}); \mathbf{var}\ \dddot{0}.x; E_D(x, 0.x, c, 0.c)) \parallel \\
 & (Q; U1(c, \ddot{m}); \mathbf{var}\ \dddot{1}.x; E_D(x, 1.x, c, 1.c)))_{+c,\ddot{m},\ddot{x}}; \hat{M}^{m,c,x} \\
= & \{\hat{M}^{m,c,x} \text{ contains an assignment to } \ddot{x};\ P,Q, \text{ separating simulations and} \\
 & \quad \text{alphabet extension do not mention } x' \text{ in their preconditions, law 2.3.49}\} \\
 & ((P; U0(c, \ddot{m}); \mathbf{var}\ \dddot{0}.x; E_D(x, 0.x, c, 0.c)) \parallel \\
 & (Q; U1(c, \ddot{m}); \mathbf{var}\ \dddot{1}.x; E_D(x, 1.x, c, 1.c)))_{+c,\ddot{m},\ddot{x}}; \mathbf{end}\ \ddot{x}; \hat{M}^{m,c,x} \\
\sqsubseteq & \{\text{Predicate calculus}\} \\
 & ((P; U0(c, \ddot{m}); \mathbf{var}\ \dddot{0}.x; E_D(x, 0.x, c, 0.c); \mathbf{end}\ \ddot{x})) \parallel \\
 & (Q; U1(c, \ddot{m}); \mathbf{var}\ \dddot{1}.x; E_D(x, 1.x, c, 1.c); \mathbf{end}\ \ddot{x}))_{+c,\ddot{m},\ddot{x}}; \hat{M}^{m,c,x} \\
= & \{\text{Lemma B.0.15, definition 2.3.101}\} \\
 & ((P; U0(c, \ddot{m}); E_D(x, x', c, 0.c); U0(\ddot{x})) \parallel \\
 & (Q; U1(c, \ddot{m}); E_D(x, x', c, 1.c); U1(\ddot{x})))_{+c,\ddot{m},\ddot{x}}; \hat{M}^{m,c,x} \\
= & \{\text{Lemma B.0.14}\} \\
 & (P_{[x]}; U0(c, \ddot{m}); U0(\ddot{x}) \parallel (Q_{[x]}; U1(c, \ddot{m}); U1(\ddot{x})))_{+c,\ddot{m},\ddot{x}}; \hat{M}^{m,c,x} \\
= & \{\text{Law 3.3.90, definition 3.3.134}\} \\
 & P_{[x]} \parallel^{m,x}_{\hat{M}} Q_{[x]}
\end{array}
$$

□

Law 3.3.148 Provided P and Q are **SH3** ∘ **S**-healthy, we have:

$$(\mathtt{var}\ x; P; \mathtt{end}\ x) \parallel^{m,c}_{\hat{M}} Q \sqsubseteq \mathtt{var}\ x; (P \parallel^{m,c,x}_{\hat{M}} Q_{[x]}); \mathtt{end}\ x$$

Proof.

$$
\begin{array}{ll}
 & (\mathtt{var}\ x; P; \mathtt{end}\ x) \parallel^{m,c}_{\hat{M}} Q \\
\sqsubseteq & \{\text{Laws 2.3.37, 2.3.41}\} \\
 & (\mathtt{var}\ x; P; \mathbf{var}\ \dddot{0}.x := \ddot{x}; \mathbf{end}\ \dddot{0}.x; \mathtt{end}\ x) \parallel^{m,c}_{\hat{M}} (Q; \mathbf{var}\ \dddot{1}.x; \mathbf{end}\ \dddot{1}.x) \\
\sqsubseteq & \{\text{Laws 3.3.70, 3.3.94, 3.3.93 and 2.3.41}\} \\
 & (\mathtt{var}\ x; P; \mathbf{var}\ \dddot{0}.x := \ddot{x}; \mathbf{end}\ \dddot{0}.x; \mathtt{end}\ x) \parallel^{m,c}_{\hat{M}} (\mathtt{var}\ x; Q_{[x]}; \mathbf{var}\ \dddot{1}.x := \ddot{x}; \mathbf{end}\ \dddot{1}.x; \mathtt{end}\ x) \\
= & \{\text{Definition 3.3.134}\} \\
 & ((\mathtt{var}\ x; P; \mathbf{var}\ \dddot{0}.x := \ddot{x}; \mathbf{end}\ \dddot{0}.x; \mathtt{end}\ x; U0(c, \ddot{m})) \parallel \\
 & \quad (\mathtt{var}\ x; Q_{[x]}; \mathbf{var}\ \dddot{1}.x := \ddot{x}; \mathbf{end}\ \dddot{1}.x; \mathtt{end}\ x; U1(c, \ddot{m})))_{+c,\ddot{m}}; \hat{M}^{m,c} \\
\sqsubseteq & \{\text{Law 3.3.93 and predicate calculus}\} \\
 & \mathtt{var}\ x; ((P; \mathbf{var}\ \dddot{0}.x := \ddot{x}; \mathbf{end}\ \dddot{0}.x; \mathbf{end}\ x; U0(c, \ddot{m})) \parallel \\
 & \quad (Q_{[x]}; \mathbf{var}\ \dddot{1}.x := \ddot{x}; \mathbf{end}\ \dddot{1}.x; \mathbf{end}\ x; U1(c, \ddot{m})))_{+c,\ddot{m}}; \mathtt{end}\ x; \hat{M}^{m,c}; \mathbf{end}\ \dddot{0}.x, \dddot{1}.x
\end{array}
$$

$$
\begin{aligned}
&= \{\text{Definition 2.3.101, law 2.3.103 (separating simulations rename all } .out \text{ variables already)}\}\\
&\quad \mathtt{var}\ x; ((P; U0(c, \dddot{m}, \ddot{x})) \parallel (Q_{[x]}; U1(c, \dddot{m}, \ddot{x})))_{+c,\dddot{m}}; \mathsf{end}\ x; \hat{M}^{m,c}; \mathbf{end}\ \dddot{0}.x, \dddot{1}.x\\
&= \{\text{Theorem 3.3.131}\}\\
&\quad \mathtt{var}\ x; ((P; U0(c, \dddot{m}, \ddot{x})) \parallel (Q_{[x]}; U1(c, \dddot{m}, \ddot{x})))_{+c,\dddot{m}}; \mathsf{end}\ x; R(m, 0.m, 1.m, m');\\
&\qquad R_{hist}(x, 0.x, 1.x, x'); c := max(0.c, 1.c); \mathbf{end}\ \dddot{0}.m, \dddot{1}.m, 0.c, 1.c; \mathbf{end}\ \dddot{0}.x, \dddot{1}.x\\
&= \{\text{Laws 3.3.93, 3.3.72}\}\\
&\quad \mathtt{var}\ x; ((P; U0(c, \dddot{m}, \ddot{x})) \parallel (Q_{[x]}; U1(c, \dddot{m}, \ddot{x})))_{+c,\dddot{m}}; R(m, 0.m, 1.m, m');\\
&\qquad R_{hist}(x, 0.x, 1.x, x'); M(x_{0.c,1.c}, (0.x \lhd 0.c \geq 1.c \rhd x_{0.c,1.c}), (1.x \lhd 1.c \geq 0.c \rhd x_{0.c,1.c}), x');\\
&\qquad \mathsf{end}\ x; c := max(0.c, 1.c); \mathbf{end}\ \dddot{0}.m, \dddot{1}.m, 0.c, 1.c; \mathbf{end}\ \dddot{0}.x, \dddot{1}.x\\
&\sqsubseteq \{\text{Laws 3.3.95, 3.3.93 and 2.3.11, definition 3.3.130}\}\\
&\quad \mathtt{var}\ x; ((P; U0(c, \dddot{m}, \ddot{x})) \parallel (Q_{[x]}; U1(c, \dddot{m}, \ddot{x})))_{+c,\dddot{m}}; R(m, 0.m, 1.m, m');\\
&\qquad R(x, 0.x, 1.x, x'); c := max(0.c, 1.c); \mathbf{end}\ \dddot{0}.m, \dddot{1}.m, 0.c, 1.c; \mathbf{end}\ \dddot{0}.x, \dddot{1}.x; \mathsf{end}\ x\\
&= \{\text{Theorem 3.3.131}\}\\
&\quad \mathtt{var}\ x; ((P; U0(c, \dddot{m}, \ddot{x})) \parallel (Q_{[x]}; U1(c, \dddot{m}, \ddot{x})))_{+c,\dddot{m}}; \hat{M}^{m,x,c}; \mathsf{end}\ x\\
&= \{\text{Definition 3.3.134}\}\\
&\quad \mathtt{var}\ x; (P \parallel^{m,c,x}_{\hat{M}} Q_{[x]}); \mathsf{end}\ x
\end{aligned}
$$

□

Law 3.3.150 $(b^{\top}_{\mathsf{S}}; b \mathbin{\overrightarrow{\ }}_{\mathsf{snc}} P) = (b^{\top}_{\mathsf{S}}; P)$

Proof.

$$
\begin{aligned}
&\quad b^{\top}_{\mathsf{S}}; b \mathbin{\overrightarrow{\ }}_{\mathsf{snc}} P\\
&= \{\text{Definition 3.3.149}\}\\
&\quad b^{\top}_{\mathsf{S}}; (P \lhd b \rhd \mathsf{sync})\\
&= \{\text{Law 3.3.58}\}\\
&\quad b^{\top}_{\mathsf{S}}; P
\end{aligned}
$$

□

Law 3.3.151 $((\neg b)^{\top}_{\mathsf{S}}; b \mathbin{\overrightarrow{\ }}_{\mathsf{snc}} P) = ((\neg b)^{\top}_{\mathsf{S}}; \mathsf{sync})$

Proof.

Similar to law 3.3.150. □

Law 3.3.152 $b_1 \mathbin{\overrightarrow{\ }}_{\mathsf{snc}} (b_2 \mathbin{\overrightarrow{\ }}_{\mathsf{snc}} P) = (b_1 \wedge b_2) \mathbin{\overrightarrow{\ }}_{\mathsf{snc}} P$

Proof.

$$
\begin{aligned}
&\quad b_1 \mathbin{\overrightarrow{\ }}_{\mathsf{snc}} (b_2 \mathbin{\overrightarrow{\ }}_{\mathsf{snc}} P)\\
&= \{\text{Definition 3.3.149}\}\\
&\quad (P \lhd b_2 \rhd \mathsf{sync}) \lhd b_1 \rhd \mathsf{sync}\\
&= \{\text{Law 3.3.22}\}
\end{aligned}
$$

$$
\begin{array}{ll}
& \mathsf{sync} \lhd \neg b_1 \rhd (\mathsf{sync} \lhd \neg b_2 \rhd P) \\
= & \{\text{Law 3.3.30}\} \\
& \mathsf{sync} \lhd \neg b_1 \vee \neg b_2 \rhd P \\
= & \{\text{Propositional calculus, law 2.3.23 and definition 3.3.149}\} \\
& (b_1 \wedge b_2) \overrightarrow{\ }_{\mathrm{snc}} P
\end{array}
$$

□

Law 3.3.153 Provided P and Q are of the form $(v \mathrel{:=}_{\mathrm{snc}} e; \mathsf{sync})$ we have:

$$b \to P \parallel_{\hat{M}} c \to Q = (b \vee c) \overrightarrow{\ }_{\mathrm{snc}} (b \to P \parallel_{\hat{M}} c \to Q)$$

Proof.

$$
\begin{array}{ll}
& (b \vee c) \overrightarrow{\ }_{\mathrm{snc}} (b \to P \parallel_{\hat{M}} c \to Q) \\
= & \{\text{Definition 3.3.149}\} \\
& ((P \lhd b \rhd \mathsf{sync}) \parallel_{\hat{M}} (Q \lhd c \rhd \mathsf{sync})) \lhd b \vee c \rhd \mathsf{sync} \\
= & \{\text{Laws 3.3.140 and 3.3.145 (proviso ensures } P \text{ and } Q \text{ are in the right form}\} \\
& (((P \parallel_{\hat{M}} Q) \lhd c \rhd P) \lhd b \rhd (Q \lhd c \rhd \mathsf{sync})) \lhd b \vee c \rhd \mathsf{sync} \\
= & \{\text{Law 3.3.30}\} \\
& (((P \parallel_{\hat{M}} Q) \lhd c \rhd P) \lhd b \rhd (Q \lhd c \rhd \mathsf{sync})) \lhd b \rhd \\
& ((((P \parallel_{\hat{M}} Q) \lhd c \rhd P) \lhd b \rhd (Q \lhd c \rhd \mathsf{sync})) \lhd c \rhd \mathsf{sync}) \\
= & \{\text{Law 3.3.26 (twice)}\} \\
& ((P \parallel_{\hat{M}} Q) \lhd c \rhd P) \lhd b \rhd (Q \lhd c \rhd \mathsf{sync}) \\
= & \{\text{Laws 3.3.140 and 3.3.145 (proviso ensures } P \text{ and } Q \text{ perform at least one } \mathsf{sync}\} \\
& (P \lhd b \rhd \mathsf{sync}) \parallel_{\hat{M}} (Q \lhd c \rhd \mathsf{sync}) \\
= & \{\text{Definition 3.3.149}\} \\
& b \to P \parallel_{\hat{M}} c \to Q
\end{array}
$$

□

Law 3.3.154 Provided $(b \wedge c = \mathbf{false})$ and P is of the form $(v \mathrel{:=}_{\mathrm{snc}} e; \mathsf{sync})$ we have:

$$b_{\mathrm{S}}^{\top}; (b \to P \parallel_{\hat{M}} c \to Q) = b_{\mathrm{S}}^{\top}; b \overrightarrow{\ }_{\mathrm{snc}} P$$

Proof.

$$
\begin{array}{ll}
& b_{\mathrm{S}}^{\top}; (b \to P \parallel_{\hat{M}} c \to Q) \\
= & \{\text{Assumption } (b \wedge c = \mathbf{false}), \text{propositional calculus}\} \\
& (b \wedge \neg c)_{\mathrm{S}}^{\top}; (b \to P \parallel_{\hat{M}} c \to Q) \\
= & \{\text{Laws 3.3.47, 3.3.143}\} \\
& b_{\mathrm{S}}^{\top}; (((\neg c)_{\mathrm{S}}^{\top}; b \overrightarrow{\ }_{\mathrm{snc}} P) \parallel_{\hat{M}} ((\neg c)_{\mathrm{S}}^{\top}; c \overrightarrow{\ }_{\mathrm{snc}} Q))
\end{array}
$$

$= \{\text{Laws 3.3.151, 3.3.143, 3.3.47 and assumption}\}$

$b_{\mathrm{S}}^{\top}; (b \rightarrow_{\mathrm{snc}} P \parallel_{\hat{M}} \mathsf{sync})$

$= \{\text{Definition 3.3.149 and law 3.3.140}\}$

$b_{\mathrm{S}}^{\top}; ((P \parallel_{\hat{M}} \mathsf{sync}) \lhd b \rhd (\mathsf{sync} \parallel_{\hat{M}} \mathsf{sync}))$

$= \{\text{Law 3.3.145 (twice), definition 3.3.149}\}$

$b_{\mathrm{S}}^{\top}; b \rightarrow_{\mathrm{snc}} P$ □

Law 3.3.155 Provided P and Q are of the form $(v \mathrel{:=}_{\mathrm{snc}} e; \mathsf{sync})$ we have:

$$b \rightarrow_{\mathrm{snc}} (P \parallel_{\hat{M}} Q) = (b \rightarrow_{\mathrm{snc}} P) \parallel_{\hat{M}} (b \rightarrow_{\mathrm{snc}} Q)$$

Proof.

$(b \rightarrow_{\mathrm{snc}} P) \parallel_{\hat{M}} (b \rightarrow_{\mathrm{snc}} Q)$

$= \{\text{Definition 3.3.149}\}$

$(P \lhd b \rhd \mathsf{sync}) \parallel_{\hat{M}} (Q \lhd b \rhd \mathsf{sync})$

$= \{\text{Laws 3.3.140 and 3.3.145 (assumption ensures proviso)}\}$

$(P \parallel_{\hat{M}} (Q \lhd b \rhd \mathsf{sync})) \lhd b \rhd (Q \lhd b \rhd \mathsf{sync})$

$= \{\text{Laws 3.3.140 and 3.3.27}\}$

$((P \parallel_{\hat{M}} Q) \lhd b \rhd (P \parallel_{\hat{M}} \mathsf{sync})) \lhd b \rhd \mathsf{sync}$

$= \{\text{Law 3.3.27 and definition 3.3.149}\}$

$b \rightarrow_{\mathrm{snc}} (P \parallel_{\hat{M}} Q)$ □

Law 3.3.156 Provided P is of the form $(v \mathrel{:=}_{\mathrm{snc}} e; \mathsf{sync})$ we have:

$$(b_1 \rightarrow_{\mathrm{snc}} P) \parallel_{\hat{M}} (b_2 \rightarrow_{\mathrm{snc}} P) = (b_1 \vee b_2) \rightarrow_{\mathrm{snc}} P$$

Proof.

$(b_1 \rightarrow_{\mathrm{snc}} P) \parallel_{\hat{M}} (b_2 \rightarrow_{\mathrm{snc}} P)$

$= \{\text{Definition 3.3.149}\}$

$(P \lhd b_1 \rhd \mathsf{sync}) \parallel_{\hat{M}} (P \lhd b_2 \rhd \mathsf{sync})$

$= \{\text{Laws 3.3.140 and 3.3.145}\}$

$((P \lhd b_1 \rhd \mathsf{sync}) \parallel_{\hat{M}} P) \lhd b_2 \rhd (P \lhd b_1 \rhd \mathsf{sync})$

$= \{\text{Laws 3.3.140 and 3.3.145, predicate calculus}\}$

$(P \lhd b_1 \rhd P) \lhd b_2 \rhd (P \lhd b_1 \rhd \mathsf{sync})$

$= \{\text{Laws 3.3.23 and 3.3.30}\}$

$P \lhd (b_1 \vee b_2) \rhd \mathsf{sync}$

= {Definition 3.3.149}

$(b_1 \vee b_2) \overrightarrow{}_{\mathsf{snc}} P$ □

Law 3.3.157 Provided P and Q are of the form $(v \mathrel{:=}_{\mathsf{snc}} e; \mathsf{sync})$ we have:

$$((s \wedge b) \overrightarrow{}_{\mathsf{snc}} P) \parallel_{\hat{M}} ((s \wedge \neg b) \overrightarrow{}_{\mathsf{snc}} Q) = s \overrightarrow{}_{\mathsf{snc}} (P \lhd b \rhd Q)$$

Proof.

$((s \wedge b) \overrightarrow{}_{\mathsf{snc}} P) \parallel_{\hat{M}} ((s \wedge \neg b) \overrightarrow{}_{\mathsf{snc}} Q)$

= {Definition 3.3.149, laws 3.3.140 and 3.3.145}

$(P \parallel_{\hat{M}} (Q \lhd (s \wedge \neg b) \rhd \mathsf{sync})) \lhd s \wedge b \rhd (Q \lhd (s \wedge \neg b) \rhd \mathsf{sync})$

= {$s \wedge b \Rightarrow ((s \wedge \neg b) = \mathbf{false}$), laws 3.3.25 and 3.3.145}

$P \lhd s \wedge b \rhd (Q \lhd (s \wedge \neg b) \rhd \mathsf{sync})$

= {Law 3.3.31}

$P \lhd s \wedge b \rhd (Q \lhd s \rhd \mathsf{sync})$

= {Law 3.3.27 and definition 3.3.149}

$s \overrightarrow{}_{\mathsf{snc}} (P \lhd b \rhd Q)$ □

Law 3.3.158 Provided $(c_{\mathsf{S}}^{\top}; P) = (c_{\mathsf{S}}^{\top}; P; c_{\mathsf{S}}^{\top})$ and that P is **S** healthy, we have:

$$c_{\mathsf{S}}^{\top}; b * P = c_{\mathsf{S}}^{\top}; (b \wedge c) * P$$

Proof.

Let:

$$F(X) = P; X \lhd b \rhd \mathrm{II}$$
$$G(X) = P; X \lhd b \wedge c \rhd \mathrm{II}$$

We first show that:

$$c_{\mathsf{S}}^{\top}; F^i(\mathbf{true}) = c_{\mathsf{S}}^{\top}; G^i(\mathbf{true})$$

by induction on i:

Case i = 0:

$c_{\mathsf{S}}^{\top}; F^0(\bot)$

= {Definition of $F^0(\bot)$}

$c_S^\top ; \bot$
= {Definition of $G^0(\bot)$}
$c_S^\top ; G^0(\mathbf{true})$ □

Case i = n+1:

$c_S^\top ; F^{n+1}(\bot)$
= {Definition of $F^i(\bot)$}
$c_S^\top ; ((P; F^n(\bot)) \lhd b \rhd \text{Ⅱ})$
= {Laws 3.3.61 and 3.3.62}
$((c_S^\top ; P; F^n(\bot)) \lhd b \wedge c \rhd (c_S^\top ; \text{Ⅱ}))$
= {Assumption}
$(c_S^\top ; P; c_S^\top ; F^n(\bot)) \lhd b \wedge c \rhd (c_S^\top ; \text{Ⅱ})$
= {Inductive hypothesis}
$(c_S^\top ; P; c_S^\top ; G^n(\bot)) \lhd b \wedge c \rhd (c_S^\top ; \text{Ⅱ})$
= {Assumption, law 3.3.62}
$c_S^\top ; ((P; G^n(\bot)) \lhd b \wedge c \rhd \text{Ⅱ})$
= {Definition of $G^{i+1}(\bot)$}
$c_S^\top ; G^{n+1}(\bot)$

With this result we can prove that $c_S^\top ; b * P = c_S^\top ; (b \wedge c) * P$:

$c_S^\top ; b * P$
= {Definition 2.3.82, definition of F, law 2.3.139 (all operators are continuous)}
$c_S^\top ; \bigsqcup_i F^i(\mathbf{true})$
= {Law 2.3.136}
$\bigsqcup_i c_S^\top ; F^i(\mathbf{true})$
= {Observation above}
$\bigsqcup_i c_S^\top ; G^i(\mathbf{true})$
= {Laws 2.3.136 and 2.3.139, definition of G}
$c_S^\top ; (b \wedge c) * P$

Law 3.3.159 $(b \underset{\text{snc}}{\rightarrow} P)_{[x]} = b \underset{\text{snc}}{\rightarrow} P_{[x]}$

Proof.

$(b \underset{\text{snc}}{\rightarrow} P)_{[x]}$

$= \{\text{Definition 3.3.149}\}$

$(P \lhd b \rhd \mathsf{sync})_{[x]}$

$= \{\text{Laws 3.3.89 and 3.3.126, definition 3.3.149}\}$

$b \overrightarrow{}_{\mathsf{snc}} P_{[x]}$ □

Law 3.3.161 Provided $i \leq k$ we have:

$$(s \,{}^{=}_{[0\,..\,i]}\, t) \wedge (s \,{}^{=}_{[0\,..\,k]}\, t) = (s \,{}^{=}_{[0\,..\,k]}\, t)$$

Proof.

Straightforward from the fact that $(s \,{}^{=}_{[0\,..\,i]}\, t)$ gets subsumed by $(s \,{}^{=}_{[0\,..\,k]}\, t)$. □

Law 3.3.162 Provided $i \leq c_0$ and $c_0 \leq k$ we have:

$$(s \,{}^{=}_{[i\,..\,k]}\, t) = (s \,{}^{=}_{[i\,..\,c_0]}\, t) \wedge (s \,{}^{=}_{[c_0\,..\,k]}\, t)$$

Proof.

Direct from the fact that $(s \,{}^{=}_{[i\,..\,c_0]}\, t)$ and $(s \,{}^{=}_{[c_0\,..\,k]}\, t)$ together account for $(s \,{}^{=}_{[i\,..\,k]}\, t)$. □

Law 3.3.163 Provided $i \leq j \leq k$ we have:

$$(s \,{}^{=}_{[i\,..\,j]}\, t) \sqsubseteq (s \,{}^{=}_{[i\,..\,k]}\, t)$$

Proof.

Straightforward from the fact that $(s \,{}^{=}_{[i\,..\,j]}\, t) \Rightarrow (s \,{}^{=}_{[i\,..\,k]}\, t)$ (as it imposes additional requirements on $s = t$) and the definition of refinement. □

Law 3.3.164 $(x \,{}^{:=}_{\mathsf{snc}}\, e; \mathsf{sync}) \wedge (x.in \,{}^{=}_{[0\,..\,c']}\, x.out') = (x \,{}^{:=}_{\mathsf{snc}}\, e; \mathsf{sync}; (x = e)^{\top}_{\mathsf{S}}) \wedge (x.in \,{}^{=}_{[0\,..\,c']}\, x.out')$

Proof.

$(x \,{}^{:=}_{\mathsf{snc}}\, e; \mathsf{sync}) \wedge (x.in \,{}^{=}_{[0\,..\,c']}\, x.out')$

$= \{\text{Laws 3.3.166 and 3.3.167}\}$

$x \,{}^{:=}_{\mathsf{snc}}\, e; \mathsf{sync} \wedge (x.in \,{}^{=}_{[0\,..\,c']}\, x.out')$

$= \{\text{Definition 3.3.124, law 3.3.16}\}$

$\mathbf{S}(\mathbf{true} \vdash c, x, x.out_c := c + 1, x.in_c, e) \wedge (x.in \,{}^{=}_{[0\,..\,c']}\, x.out')$

$= \{\text{Definition 2.3.1, propositional calculus}\}$

$\mathbf{S}(\mathbf{true} \vdash c, x, x.out_c := c + 1, x.in_c, e \wedge (x.in \,{}^{=}_{[0\,..\,c']}\, x.out')) \wedge (x.in \,{}^{=}_{[0\,..\,c']}\, x.out')$

$= \{\text{Propositional calculus, definition 2.3.1}\}$

$\mathbf{S}(\mathbf{true} \vdash c, x, x.out_c := c + 1, e, e) \wedge (x.in \,{}^{=}_{[0\,..\,c']}\, x.out')$

$= \{\text{Definition of assignment, law 3.3.53}\}$

$(\mathbf{S}(\mathbf{true} \vdash c, x, x.out_c := c + 1, e, e); (x = e)^{\top}_{\mathsf{S}}) \wedge (x.in \,{}^{=}_{[0\,..\,c']}\, x.out')$

= {Law 3.3.166, definition 2.3.1, propositional calculus}

$\mathbf{S}(\mathbf{true} \vdash (c, x, x.out_c := c + 1, e, e) \wedge (x.in_{[0\,..\,c']}^{=} x.out')) \wedge (x.in_{[0\,..\,c']}^{=} x.out');$

$(x = e)_{\mathsf{S}}^{\top} \wedge (x.in_{[0\,..\,c']}^{=} x.out')$

= {Propositional calculus, definition 2.3.1 and law 3.3.166}

$(\mathbf{S}(\mathbf{true} \vdash c, x, x.out_c := c + 1, x.in_c, e); (x = e)_{\mathsf{S}}^{\top}) \wedge (x.in_{[0\,..\,c']}^{=} x.out')$

= {Law 3.3.16, definition 3.3.124}

$(x \overset{:=}{_{\mathsf{snc}}} e; \mathsf{sync}; (x = e)_{\mathsf{S}}^{\top}) \wedge (x.in_{[0\,..\,c']}^{=} x.out')$ □

Law 3.3.165 $(P \lhd b \rhd Q) \wedge (x.in_{[0\,..\,c']}^{=} x.out') = (P \wedge (x.in_{[0\,..\,c']}^{=} x.out')) \lhd b \rhd (Q \wedge (x.in_{[0\,..\,c']}^{=} x.out'))$

Proof.

$(P \lhd b \rhd Q) \wedge (x.in_{[0\,..\,c']}^{=} x.out')$

= {Definition 2.2.1}

$((b \wedge P) \vee (\neg b \wedge Q)) \wedge (x.in_{[0\,..\,c']}^{=} x.out')$

= {Propositional calculus}

$(b \wedge P \wedge (x.in_{[0\,..\,c']}^{=} x.out')) \vee (\neg b \wedge Q \wedge (x.in_{[0\,..\,c']}^{=} x.out'))$

= {Definition 2.2.1}

$(P \wedge (x.in_{[0\,..\,c']}^{=} x.out')) \lhd b \rhd (Q \wedge (x.in_{[0\,..\,c']}^{=} x.out'))$ □

Law 3.3.166 Provided P and Q are $\mathbf{S}$-healthy we have:

$(P; Q) \wedge (x.in_{[0\,..\,c']}^{=} x.out') = P \wedge (x.in_{[0\,..\,c']}^{=} x.out'); Q \wedge (x.in_{[0\,..\,c']}^{=} x.out')$

Proof.

$P \wedge (x.in_{[0\,..\,c']}^{=} x.out'); Q \wedge (x.in_{[0\,..\,c']}^{=} x.out')$

= {Assumption (P and Q are $\mathbf{S}$))}

$P \wedge (x.in' = x.in) \wedge (x.in_{[0\,..\,c']}^{=} x.out'); Q \wedge (c \leq c') \wedge (x.out \leq x.out') \wedge (x.in_{[0\,..\,c']}^{=} x.out')$

= {Definition 2.2.3}

$\exists v_0, x_0, x.in_0, x.out_0, c_0 \bullet$

$P[v_0, x_0, x.out_0, c_0/v', x', x.out', c'] \wedge (x.in_0 = x.in) \wedge (x.in_{[0\,..\,c_0]}^{=} x.out_0) \wedge$

$Q[v_0, x_0, x.out_0, x.in_0, c_0/v, x, x.out, x.in, c] \wedge$

$(c_0 \leq c') \wedge (x.out_0 \leq x.out') \wedge (x.in_{0\,[0\,..\,c']}^{=} x.out')$

= {Propositional calculus, law 3.3.161, quantifier contract scope}

$(\exists v_0, x_0, x.in_0, x.out_0, c_0 \bullet P[v_0, x_0, x.out_0, c_0/v', x', x.out', c'] \wedge (x.in_0 = x.in) \wedge$

$Q[v_0, x_0, x.out_0, x.in_0, c_0/v, x, x.out, x.in, c] \wedge$

$(c_0 \leq c') \wedge (x.out_0 \leq x.out')) \wedge (x.in_{[0\,..\,c']}^{=} x.out')$

$$
\begin{array}{ll}
& = \{\text{Definition 2.2.3}\} \\
& ((P \wedge (x.in' = x.in)); (Q \wedge (c \leq c') \wedge (x.out \leq x.out'))) \wedge (x.in_{[0 \,..\, c']}^{=} x.out') \\
& = \{\text{Assumption } (P \text{ and } Q \text{ are } \mathbf{S})\} \\
& (P; Q) \wedge (x.in_{[0 \,..\, c']}^{=} x.out') \qquad \square
\end{array}
$$

Law 3.3.167 Provided P and Q are **S**-healthy we have:

$$P; Q \wedge (x.in_{[0 \,..\, c']}^{=} x.out') = P \wedge (x.in_{[0 \,..\, c']}^{=} x.out'); Q \wedge (x.in_{[0 \,..\, c']}^{=} x.out')$$

Proof.

$$
\begin{array}{ll}
& P \wedge (x.in_{[0 \,..\, c']}^{=} x.out'); Q \wedge (x.in_{[0 \,..\, c']}^{=} x.out') \\
& = \{\text{Assumption } (P \text{ and } Q \text{ are } \mathbf{S})\} \\
& P \wedge (x.in' = x.in) \wedge (x.in_{[0 \,..\, c']}^{=} x.out'); Q \wedge (c \leq c') \wedge (x.out \leq x.out') \wedge (x.in_{[0 \,..\, c']}^{=} x.out') \\
& = \{\text{Definition 2.2.3}\} \\
& \exists v_0, x_0, x.in_0, x.out_0, c_0 \bullet \\
& \quad P[v_0, x_0, x.out_0, c_0 / v', x', x.out', c'] \wedge (x.in_0 = x.in) \wedge (x.in_{[0 \,..\, c_0]}^{=} x.out_0) \wedge \\
& \quad Q[v_0, x_0, x.out_0, x.in_0, c_0 / v, x, x.out, x.in, c] \wedge \\
& \quad (c_0 \leq c') \wedge (x.out_0 \leq x.out') \wedge (x.in_0{}_{[0 \,..\, c']}^{=} x.out') \\
& = \{\text{Law 3.3.161}\} \\
& (\exists v_0, x_0, x.in_0, x.out_0, c_0 \bullet P[v_0, x_0, x.out_0, c_0 / v', x', x.out', c'] \wedge (x.in_0 = x.in) \wedge \\
& \quad Q[v_0, x_0, x.out_0, x.in_0, c_0 / v, x, x.out, x.in, c] \wedge \\
& \quad (c_0 \leq c') \wedge (x.out_0 \leq x.out') \wedge (x.in_0{}_{[0 \,..\, c']}^{=} x.out')) \\
& = \{\text{Definition 2.2.3}\} \\
& (P \wedge (x.in' = x.in)); (Q \wedge (c \leq c') \wedge (x.out \leq x.out') \wedge (x.in_{[0 \,..\, c']}^{=} x.out')) \\
& = \{\text{Assumption } (P \text{ and } Q \text{ are } \mathbf{S})\} \\
& P; Q \wedge (x.in_{[0 \,..\, c']}^{=} x.out') \qquad \square
\end{array}
$$

Law 3.3.168 Provided P is **SH3** and a **S**-healthy design, we have:

$$P; \mathbb{II} \wedge (x.in_{[0 \,..\, c']}^{=} x.out') = P \wedge (x.in_{[0 \,..\, c']}^{=} x.out')$$

Proof.

$$
\begin{array}{ll}
& P; \mathbb{II} \wedge (x.in_{[0 \,..\, c']}^{=} x.out') \\
& = \{\text{Law 3.3.167}\} \\
& P \wedge (x.in_{[0 \,..\, c']}^{=} x.out'); \mathbb{II} \wedge (x.in_{[0 \,..\, c']}^{=} x.out') \\
& = \{\text{Law 3.3.166}\}
\end{array}
$$

$$
\begin{array}{ll}
& (P; \Pi) \wedge (x.in \underset{[0..c']}{=} x.out') \\
= & \{\text{Theorem 3.3.10}\} \\
& P \wedge (x.in \underset{[0..c']}{=} x.out')
\end{array}
$$

□

Law 3.3.169 Provided P is **S**-healthy we have:

$$((b * P) \wedge (x.in \underset{[0..c']}{=} x.out')) = b * (P \wedge (x.in \underset{[0..c']}{=} x.out'))$$

Proof. Let $F = P \lhd b \rhd \Pi$, then we have:

$$
\begin{array}{ll}
& (b * P) \wedge (x.in \underset{[0..c']}{=} x.out')) \\
= & \{\text{Definition 2.3.82, definition of } F \text{ above and law 2.3.139 (all operators are continuous)}\} \\
& \left(\bigsqcup_i F^i(\bot)\right) \wedge (x.in \underset{[0..c']}{=} x.out') \\
= & \{\text{Laws 3.3.168 and 2.3.135}\} \\
& \bigsqcup_i (F^i(\bot) \wedge (x.in \underset{[0..c']}{=} x.out')) \\
= & \{\text{Law 2.3.139, definition of } F \text{ and definition 2.3.82}\} \\
& b * (P \wedge (x.in \underset{[0..c']}{=} x.out'))
\end{array}
$$

□

Law 3.3.170 Provided P is **S** and neither P nor **S** mention x we have:

$$((x = e)^{\top}_{\mathsf{S}}; P_{[x]}) \wedge (x.in \underset{[0..c']}{=} x.out') = ((x = e)^{\top}_{\mathsf{S}}; P_{[x]}; (x = e)^{\top}_{\mathsf{S}}) \wedge (x.in \underset{[0..c']}{=} x.out')$$

In order to prove this result, we first show three lemmas that help structuring the proof.

Lemma B.0.19.

$$(\exists x_0 \bullet x = e \wedge E(x, c_0)[x_0, 0.x.out/x', x.out'] \wedge (\neg x_0 = e) \wedge (x.in \underset{[0..0.c]}{=} 0.x.out)) = \textbf{\textit{false}}$$

Proof.

$$
\begin{array}{ll}
& \exists x_0 \bullet x = e \wedge E(x, c_0)[x_0, 0.x.out/x', x.out'] \wedge (\neg x_0 = e) \wedge (x.in \underset{[0..0.c]}{=} 0.x.out) \\
= & \{\text{Definition of } E(x, c') \text{ then case analysis on } c = c_0\}
\end{array}
$$

Case $c = c_0$

$$
\begin{array}{l}
\exists x_0 \bullet x = e \wedge (x.in \underset{[0..0.c]}{=} 0.x.out) \wedge (x_0, 0.x.out_c, x.in' = (x.in_{c_0-1}, x) \lhd c < c_0 \rhd (x, x.out_c), x.in) \wedge \\
\qquad \left\{0.x.out'_i = x.in_{i-1} \mid c < i < c_0\right\} \wedge \left\{0.x.out'_i = x.out_i \mid i < c\right\} \wedge \neg(x_0 = e)
\end{array}
$$

$$
\begin{array}{ll}
= & \{\text{Case } c = c_0, \text{ propositional calculus}\} \\
& \exists x_0 \bullet x = e \wedge (x.in \underset{[0..0.c]}{=} 0.x.out) \wedge (x_0, 0.x.out_c, x.in' = x, x.out_c, x.in) \wedge \neg(x_0 = e) \wedge
\end{array}
$$

$$\left\{0.x.out'_i = x.out_i \mid i < c\right\}$$

= {Contradiction: $x_0 = e$ and $\neg(x_0 = e)$}

false

Case $c < c_0$:

$$\exists x_0 \bullet x = e \wedge (x.in_{[0\,..\,\overset{=}{0.c]}} 0.x.out) \wedge (x_0, 0.x.out_c, x.in' = (x.in_{c_0-1}, x) \lhd c < c_0 \rhd (x, x.out_c), x.in) \wedge$$
$$\left\{0.x.out'_i = x.in_{i-1} \mid c < i < c_0\right\} \wedge \left\{0.x.out'_i = x.out_i \mid i < c\right\} \wedge \neg(x_0 = e)$$

= {Case $c < c'$, propositional calculus}

$$\exists x_0 \bullet x = e \wedge (x.in_{[0\,..\,\overset{=}{0.c]}} 0.x.out) \wedge (x_0, 0.x.out_c, x.in' = x.in_{c_0-1}, e, x.in) \wedge$$
$$\left\{0.x.out'_i = x.in_{i-1} \mid c < i < c_0\right\} \wedge \left\{0.x.out'_i = x.out_i \mid i < c\right\} \wedge \neg(x_0 = e)$$

= {Propositional calculus}

$$\exists x_0 \bullet x = e \wedge (x.in_{[0\,..\,\overset{=}{0.c]}} 0.x.out) \wedge (x_0, x.in' = x.in_{c_0-1}, x.in) \wedge$$
$$0.x.out = x.out_{0\ldots c-1} \frown \langle e \rangle \frown \langle x.in_c, \ldots, x.in_{c_0-1} \rangle \wedge \neg(x_0 = e)$$

= {Propositional calculus ($x.in_{[0\,..\,\overset{=}{c_0]}} 0.x.out$)}

$$\exists x_0 \bullet x = e \wedge (x.in_{[0\,..\,\overset{=}{0.c]}} 0.x.out) \wedge (x_0, x.in' = x.in_{c_0-1}, x.in) \wedge$$
$$0.x.out = x.out_{0\ldots c-1} \frown \langle e \rangle \frown \langle 0.x.out_c, \ldots, 0.x.out_{c_0-1} \rangle \wedge \neg(x_0 = e)$$

= {Propositional calculus ($x.in_{c'-1} = x.out_{c'-1}$)}

$$\exists x_0 \bullet x = e \wedge (x.in_{[0\,..\,\overset{=}{0.c]}} 0.x.out) \wedge (x_0, x.in' = x.in_{c_0-1}, x.in) \wedge$$
$$0.x.out = x.out_{0\ldots c-1} \frown \langle e \rangle \frown \langle e, \ldots, e \rangle \wedge \neg(x_0 = e)$$

= {$(x_0 = x.in_{c_0-1}) \wedge (x.in_{c_0-1} = 0.x.out_{c_0-1}) \Rightarrow x_0 = e$ (contradiction)}

false □

Lemma B.0.20. *Provided neither P nor* **S** *mention x we have:*

$$(x = e \wedge (\mathbf{S}(P_2 \wedge E(x, c')); \mathbf{S}(\neg x = e)) \wedge (x.in_{[0\,..\,\overset{=}{c'}]} x.out')) = \textbf{\textit{false}}$$

Proof.

$$(x = e \wedge (\mathbf{S}(P_2 \wedge E(x, c')); \mathbf{S}(\neg x = e)) \wedge (x.in_{[0\,..\,\overset{=}{c'}]} x.out')$$

= {Definition 2.2.3, assumption}

$$(x = e \wedge (x.in_{[0\,..\,\overset{=}{c'}]} x.out') \wedge$$
$$(\exists x_0, c_0, v_0 \bullet P_2[c_0, v_0/c', v'] \wedge E(x, c')[x_0, c_0/v_0, c'] \wedge$$
$$(\neg x_0 = e) \wedge \mathbf{S}[c_0, v_0/c', v'] \wedge \mathbf{S}[c_0, v_0/c, v])$$

= {Propositional calculus}

$$x = e \wedge (x.in_{[0\,..\,\overset{=}{c'}]} x.out') \wedge (\exists c_0, x.out_0, v_0 \bullet P_2[c_0, v_0, x.out_0/c', v', x.out'] \wedge$$
$$\mathbf{S}[c_0, v_0, x.out_0/c', v', x.out'] \wedge \mathbf{S}[c_0, v_0, x.out_0/c, v, x.out] \wedge$$

$$(\exists x_0 \bullet x = e \wedge E(x, c_0)[x_0, x.out_0/x', x.out'] \wedge (\neg x_0 = e) \wedge (x.in_{[0\,..\,0.c]}^{=} 0.x.out)))$$

= {Lemma B.0.19 and propositional calculus}

false □

Lemma B.0.21.

$$(x = e \wedge P_{[x]} \wedge (x.in_{[0\,..\,c']}^{=} x.out')) = (x = e \wedge (P_{[x]}; (x = e)_S^{\top}) \wedge (x.in_{[0\,..\,c']}^{=} x.out'))$$

Proof.

$$(x = e \wedge (P_{[x]}; (x = e)_S^{\top}) \wedge (x.in_{[0\,..\,c']}^{=} x.out'))$$

= {Theorem 3.3.6, definitions 2.3.1 and 3.3.82, propositional calculus}

$$x = e \wedge \mathbf{S}(\mathbf{S}(\neg ok) \vee \mathbf{S}(\neg P_1); \mathbf{S}(\mathbf{true}) \vee (\mathbf{S}(P_2 \wedge E(x, c')); \mathbf{S}(\neg x = e)) \vee$$
$$(ok' \wedge \mathbf{S}(P_2 \wedge E(x, c')); \mathbf{S}(\mathbb{I}))) \wedge (x.in_{[0\,..\,c']}^{=} x.out')$$

= {Theorem 3.2.9, law 2.2.8 and propositional calculus}

$$x = e \wedge \mathbf{S}(\mathbf{S}(\neg ok) \vee \mathbf{S}(\neg P_1); \mathbf{S}(\mathbf{true}) \vee (ok' \wedge \mathbf{S}(P_2 \wedge E(x, c'))) \vee$$
$$(x = e \wedge (\mathbf{S}(P_2 \wedge E(x, c')); \mathbf{S}(\neg x = e)) \wedge (x.in_{[0\,..\,c']}^{=} x.out'))) \wedge (x.in_{[0\,..\,c']}^{=} x.out')$$

= {Lemma B.0.20 and propositional calculus}

$$x = e \wedge \mathbf{S}(\mathbf{S}(\neg ok) \vee \mathbf{S}(\neg P_1); \mathbf{S}(\mathbf{true}) \vee (ok' \wedge \mathbf{S}(P_2 \wedge E(x, c')))) \wedge (x.in_{[0\,..\,c']}^{=} x.out')$$

= {Theorem 3.2.14, propositional calculus, definition 2.3.1}

$$x = e \wedge \mathbf{S}(P_1 \vdash P_2 \wedge E(x, c')) \wedge (x.in_{[0\,..\,c']}^{=} x.out')$$

= {Definition 3.3.82, P is $\mathbf{S}$}

$$x = e \wedge P_{[x]} \wedge (x.in_{[0\,..\,c']}^{=} x.out')$$ □

With these results we can now prove the result we wanted:

$$((x = e)_S^{\top}; P_{[x]}) \wedge (x.in_{[0\,..\,c']}^{=} x.out') = ((x = e)_S^{\top}; P_{[x]}; (x = e)_S^{\top}) \wedge (x.in_{[0\,..\,c']}^{=} x.out')$$

Proof.

$$((x = e)_S^{\top}; P_{[x]}) \wedge (x.in_{[0\,..\,c']}^{=} x.out')$$

= {Definition 3.3.34, laws 3.3.33, 3.3.7, 3.3.9 and 3.3.165}

$$(P_{[x]} \wedge (x.in_{[0\,..\,c']}^{=} x.out')) \lhd x = e \rhd (\top \wedge (x.in_{[0\,..\,c']}^{=} x.out'))$$

= {Definition 2.2.1, lemma B.0.21}

$$((P_{[x]}; (x = e)_S^{\top}) \wedge (x.in_{[0\,..\,c']}^{=} x.out')) \lhd x = e \rhd (\top \wedge (x.in_{[0\,..\,c']}^{=} x.out'))$$

= {Laws 3.3.165, 3.3.7, 3.3.9 and 3.3.33}

$$((\mathbb{I} \lhd x = e \rhd \top); P_{[x]}; (x = e)_S^{\top}) \wedge (x.in_{[0\,..\,c']}^{=} x.out')$$

= {Definition 3.3.34}

$$((x = e)^{\top}_{\mathsf{S}}; P_{[x]}; (x = e)^{\top}_{\mathsf{S}}) \wedge (x.in_{[0\,..\,c']}^{=} x.out')$$ □

Law 3.3.171 Provided P is **H3** and **S** we have:

$$((x = e)^{\top}; (b * P)_{[x]}) \wedge (x.in_{[0\,..\,c']}^{=} x.out') \sqsubseteq ((x = e)^{\top}; (b \wedge x = e) * P_{[x]}) \wedge (x.in_{[0\,..\,c']}^{=} x.out')$$

Proof.

$$\begin{array}{ll}
 & ((x = e)^{\top}; (b * P)_{[x]}) \wedge (x.in_{[0\,..\,c']}^{=} x.out') \\
\sqsubseteq & \{\text{Laws 3.3.91, 3.3.166 and 3.3.167}\} \\
 & (x = e)^{\top}; (b * P_{[x]}) \wedge (x.in_{[0\,..\,c']}^{=} x.out') \\
= & \{\text{Laws 3.3.169 and 3.3.158 (law 3.3.170 ensures proviso)}\} \\
 & (x = e)^{\top}; (b \wedge x = e) * P_{[x]} \wedge (x.in_{[0\,..\,c']}^{=} x.out') \\
= & \{\text{Laws 3.3.169, 3.3.167 and 3.3.166}\} \\
 & ((x = e)^{\top}; (b \wedge x = e) * P_{[x]}) \wedge (x.in_{[0\,..\,c']}^{=} x.out')
\end{array}$$ □

Appendix C

Proofs from Chapter 4

Unless stated otherwise, the proofs of the laws in this section are a direct consequence of the semantics described in section 4.1 and the corresponding property in the underlying operator in the synchronous theory.

Law 4.1.13 $P \fatsemi (Q \fatsemi S) = (P \fatsemi Q) \fatsemi S$

Law 4.1.14 $(\mathbb{I} \fatsemi P) \sqsubseteq P$

Law 4.1.15 $(\bot \fatsemi P) = \bot$

Law 4.1.16 $P \parallel_{HC} Q = Q \parallel_{HC} P$

Law 4.1.17 $P \parallel_{HC} (Q \parallel_{HC} S) = (P \parallel_{HC} Q) \parallel_{HC} S$

Law 4.1.18 $(P \parallel_{HC} \mathbb{I}) = P$

Law 4.1.19 $(P \parallel_{HC} \bot) = \bot$

Law 4.1.20 $x :=_{HC} e \fatsemi (P \parallel_{HC} Q) = (x :=_{HC} e \fatsemi P) \parallel_{HC} (\mathbf{delay} \fatsemi Q)$

Law 4.1.21 $x, y :=_{HC} e_1, e_2 \fatsemi (P \parallel_{HC} Q) = (x :=_{HC} e_1 \fatsemi P) \parallel_{HC} (y :=_{HC} e_2 \fatsemi Q)$

Law 4.1.22

$$(ch?x \fatsemi P) \parallel_{HC} (ch!e \fatsemi Q) = (x, ch?, ch!, ch :=_{HC} e, true, true, e) \fatsemi (P \parallel_{HC} Q)$$

Law 4.1.23

$$(ch?x \fatsemi P) \parallel_{HC} (ch!e \fatsemi Q) \parallel_{HC} (ch?y \fatsemi R) = (x, y, ch?, ch!, ch :=_{HC} e, e, true, true, e) \fatsemi (P \parallel_{HC} Q \parallel_{HC} R)$$

Law 4.1.24

$$[\![\mathbf{priAlt} \{ \mathbf{case}\ ch_1?x\mathbf{:}\ P_1 \fatsemi \mathbf{break} \fatsemi ; \langle \mathsf{guard\_list} \rangle \fatsemi ; \mathbf{default:}\ P_n \}]\!] = [\![\mathbf{priAlt} \{ \mathbf{case}\ ch_1?x\mathbf{:}\ P_1 \fatsemi \mathbf{break} \fatsemi ; \mathbf{default:}\ [\![\mathbf{priAlt} \{ \langle \mathsf{guard\_list} \rangle \fatsemi ; \mathbf{default:}\ P_n \}]\!] \}]\!]$$

We begin by proving that a priAlt command with a default guard does not iterate. This is not a novel result as priAlts with default clauses are defined to not to iterate. The importance of this result is that we can actually show this behaviour from the semantics.

Lemma C.0.22. *Provided that P is a list of case statements, we have that:*

$$[\![\mathbf{priAlt} \{ \mathbf{case}\ ch_1?x\mathbf{:}\ P_1 \fatsemi \mathbf{break} \fatsemi ; \mathbf{default:}\ P_2 \}]\!] = ch_1?\ :=_{snc}\ \mathbf{true}; m :=_{snc} ch_1.in_c; \mathbf{sync}; P_1 \lhd ch_1!.in_c \rhd P_2$$

Proof.

For the proof let's consider $P = \mathbf{case}\ ch_1?x\mathbf{:}\ P_1 \fatsemi \mathbf{break} \fatsemi \mathbf{default:}\ P_2$. priAlts with deeper case lists can be proved in a similar way.

$[\![\mathbf{priAlt}\, \{P\}]\!]$

= {Assumption, definitions 4.1.9, 4.1.10 and 4.1.12}

$\mathbf{var\ res} :=_{snc} \mathbf{false}; \mu X \bullet$

$\quad \mathbb{I} \lhd \mathbf{res} \rhd (ch?\ :=_{snc} \mathbf{true}; (\mathbf{res}, m :=_{snc} \mathbf{true}, ch.in_c; \mathbf{sync}; P_{1+\mathbf{res}} \lhd ch!.in_c \rhd \mathbb{I}));$

$\quad \mathbb{I} \lhd \mathbf{res} \rhd (\mathbf{res} :=_{snc} \mathbf{true}; P_{2+\mathbf{res}});$

$\quad \mathbb{I} \lhd \mathbf{res} \rhd \mathsf{sync}; X;$

$\mathbf{end\ res}$

= {Law 2.3.80}

$\mathbf{var\ res} :=_{snc} \mathbf{false};$

$\quad \mathbb{I} \lhd \mathbf{res} \rhd (ch?\ :=_{snc} \mathbf{true}; (\mathbf{res}, m :=_{snc} \mathbf{true}, ch.in_c; \mathbf{sync}; P_{1+\mathbf{res}} \lhd ch!.in_c \rhd \mathbb{I}));$

$\quad \mathbb{I} \lhd \mathbf{res} \rhd (\mathbf{res} :=_{snc} \mathbf{true}; P_{2+\mathbf{res}});$

$\quad \mathbb{I} \lhd \mathbf{res} \rhd \mathsf{sync}; [\![\mathbf{priAlt}\, \{P\}]\!];$

$\mathbf{end\ res}$

= {Laws 3.3.32 and 3.3.25}

$\mathbf{var\ res};$

$\quad \mathbf{res} :=_{snc} \mathbf{false}; (ch?\ :=_{snc} \mathbf{true}; (\mathbf{res}, m :=_{snc} \mathbf{true}, ch.in_c; \mathbf{sync}; P_{1+\mathbf{res}} \lhd ch!.in_c \rhd \mathbb{I}));$

$\quad \mathbb{I} \lhd \mathbf{res} \rhd (\mathbf{res} :=_{snc} \mathbf{true}; P_{2+\mathbf{res}});$

$\quad \mathbb{I} \lhd \mathbf{res} \rhd \mathsf{sync}; [\![\mathbf{priAlt}\, \{P\}]\!];$

$$
\begin{array}{l}
\quad \textbf{end res} \\
= \{\text{Laws 3.3.32 and 3.3.16 and predicate calculus}\} \\
\quad \textbf{var res}; ch?\ \underset{\text{snc}}{:=}\ \textbf{true}; \\
\qquad (m\ \underset{\text{snc}}{:=}\ ch.in_c; \textbf{sync}; P_{1+\textbf{res}}; \textbf{res}\ \underset{\text{snc}}{:=}\ \textbf{true} \lhd ch!.in_c \rhd \textbf{res}\ \underset{\text{snc}}{:=}\ \textbf{false}); \\
\qquad \amalg \lhd \textbf{res} \rhd (\textbf{res}\ \underset{\text{snc}}{:=}\ \textbf{true}; P_{2+\textbf{res}}); \\
\qquad \amalg \lhd \textbf{res} \rhd \mathsf{sync}; [\![\textbf{priAlt}\,\{P\}]\!]\,; \\
\quad \textbf{end res} \\
= \{\text{Laws 3.3.33, 3.3.32 and 3.3.25, predicate calculus}\} \\
\quad \textbf{var res}; ch?\ \underset{\text{snc}}{:=}\ \textbf{true}; \\
\qquad (m\ \underset{\text{snc}}{:=}\ ch.in_c; \textbf{sync}; P_{1+\textbf{res}}; \textbf{res}\ \underset{\text{snc}}{:=}\ \textbf{true} \lhd ch!.in_c \rhd (P_{2+\textbf{res}}; \textbf{res}\ \underset{\text{snc}}{:=}\ \textbf{true})); \\
\qquad \amalg \lhd \textbf{res} \rhd \mathsf{sync}; [\![\textbf{priAlt}\,\{P\}]\!]\,; \\
\quad \textbf{end res} \\
= \{\text{Laws 3.3.33, 3.3.32 and 3.3.16}\} \\
\quad \textbf{var res}; (ch?\ \underset{\text{snc}}{:=}\ \textbf{true}; m\ \underset{\text{snc}}{:=}\ ch.in_c; \textbf{sync}; P_{1+\textbf{res}} \lhd ch!.in_c \rhd P_{2+\textbf{res}}); \textbf{res}\ \underset{\text{snc}}{:=}\ \textbf{true}; \textbf{end res} \\
= \{\text{Laws 2.3.40, predicate calculus and definitions 4.1.10 and 4.1.12}\} \\
\quad ch?\ \underset{\text{snc}}{:=}\ \textbf{true}; m\ \underset{\text{snc}}{:=}\ ch.in_c; \textbf{sync}; P_1 \lhd ch!.in_c \rhd P_2 \qquad \square
\end{array}
$$

We can now prove law 4.1.24:

Proof.

$$
\begin{array}{l}
\quad [\![\textbf{priAlt}\,\{\textbf{case}\ ch_1?x\textbf{:}\ P_1 \fatsemi \textbf{break} \fatsemi \textbf{case}\ ch_2?y\textbf{:}\ P_2 \fatsemi \textbf{break} \fatsemi \textbf{default:}\ P_3\}]\!] \\
= \{\text{Lemma C.0.22}\} \\
\quad ch_1?\ \underset{\text{snc}}{:=}\ \textbf{true}; (x\ \underset{\text{snc}}{:=}\ ch_1.in_c; \textbf{sync}; P_1) \lhd ch_1!.in_c \rhd \\
\qquad (ch_2?\ \underset{\text{snc}}{:=}\ \textbf{true}; (y\ \underset{\text{snc}}{:=}\ ch_2.in_c; \textbf{sync}; P_2) \lhd ch_2!.in_c \rhd P_3) \\
= \{\text{Lemma C.0.22}\} \\
\quad ch_1?\ \underset{\text{snc}}{:=}\ \textbf{true}; (x\ \underset{\text{snc}}{:=}\ ch_1.in_c; \textbf{sync}; P_1) \lhd ch_1!.in_c \rhd \\
\qquad [\![\textbf{priAlt}\,\{\textbf{case}\ ch_2?y\textbf{:}\ P_2 \fatsemi \textbf{break} \fatsemi \textbf{default:}\ P_3\}]\!] \\
= \{\text{Lemma C.0.22}\} \\
\quad [\![\textbf{priAlt}\,\{\textbf{case}\ ch_1?x\textbf{:}\ P_1 \fatsemi \textbf{break} \fatsemi \textbf{default:}\ [\![\textbf{priAlt}\,\{\langle \mathsf{guard\_list} \rangle \fatsemi \textbf{default:}\ P_n\}]\!]\}]\!] \qquad \square
\end{array}
$$

Law 4.1.25 Provided that P is a **SH3**, **S**-healthy design and it does not mention **res** we have that:

$$\textbf{priAlt}\,\{\textbf{case}\ ch?x\textbf{:}\ P \fatsemi \textbf{break}\} = ch?x \fatsemi P$$

Proof.

Let

$F(X) = \mathbb{I} \lhd \mathbf{res} \rhd (ch?, \mathbf{res}, m \mathrel{:=}_{snc} \mathbf{true}, \mathbf{true}, ch.in_c; \mathsf{sync}; P \lhd ch!.in_c \rhd ch? \mathrel{:=}_{snc} \mathbf{true}; \mathsf{sync}; X)$

$G(X) = ch?, m \mathrel{:=}_{snc} \mathbf{true}, ch.in_c; \mathsf{sync}; P \lhd ch!.in_c \rhd ch? \mathrel{:=}_{snc} \mathbf{true}; \mathsf{sync}; X$

We begin by observing that:

$\mathbf{var\ res} := \mathbf{false}; F^n(\bot); \mathbf{end\ res} = G^n(\bot)$

We show this result by induction on n:
Base case (n = 0):

$\mathbf{var\ res} := \mathbf{false}; F^0(\bot); \mathbf{end\ res}$
$= \{F^0(\bot) = \bot$, law 3.3.8 and predicate calculus$\}$
$\bot$
$= \{G^0(\bot) = \bot\}$
$G^0(\bot)$

Inductive hypothesis: $\mathbf{var\ res} := \mathbf{false}; F^i(\bot); \mathbf{end\ res} = G^i(\bot)$
Inductive step (n = i+1):

$\mathbf{var\ res} := \mathbf{false}; F^{i+1}(\bot); \mathbf{end\ res}$
$= \{$Definition of $F^n(\bot)\}$
$\mathbf{var\ res} := \mathbf{false};$
$\mathbb{I} \lhd \mathbf{res} \rhd (ch?, \mathbf{res}, m \mathrel{:=}_{snc} \mathbf{true}, \mathbf{true}, ch.in_c; \mathsf{sync}; P \lhd ch!.in_c \rhd ch? \mathrel{:=}_{snc} \mathbf{true}; \mathsf{sync}; F^i(\bot));$
$\mathbf{end\ res}$
$= \{$Laws 3.3.32, 3.3.16, 3.3.25$\}$
$\mathbf{var\ res};$
$(ch?, \mathbf{res}, m \mathrel{:=}_{snc} \mathbf{true}, \mathbf{true}, ch.in_c; \mathsf{sync}; P \lhd ch!.in_c \rhd \mathbf{res}, ch? \mathrel{:=}_{snc} \mathbf{false}, \mathbf{true}; \mathsf{sync}; F^i(\bot));$
$\mathbf{end\ res}$
$= \{$Definitions 2.2.18, 2.2.19 and 2.2.1, propositional calculus$\}$
$(\mathbf{var\ res}; ch?, \mathbf{res}, m \mathrel{:=}_{snc} \mathbf{true}, \mathbf{true}, ch.in_c; \mathsf{sync}; P; \mathbf{end\ res})$
$\lhd ch!.in_c \rhd$
$(\mathbf{var\ res}; \mathbf{res}, ch? \mathrel{:=}_{snc} \mathbf{false}, \mathbf{true}; \mathsf{sync}; F^i(\bot); \mathbf{end\ res})$
$= \{$Laws 3.3.17, 2.3.47, 2.3.48 and 2.3.40$\}$
$(ch?, m \mathrel{:=}_{snc} \mathbf{true}, ch.in_c; \mathsf{sync}; \mathbf{var\ res}; \mathbf{end\ res}; P)$

$$\lhd\ ch!.in_c\ \rhd$$

$(ch?\ \mathrel{:=}_{\mathrm{snc}}\ \mathbf{true};\ \mathtt{sync};\ \mathbf{var\ res};\ \mathbf{res}\ \mathrel{:=}_{\mathrm{snc}}\ \mathbf{false};\ F^i(\bot);\ \mathbf{end\ res})$

= {Inductive hypothesis, laws 3.3.70 and 3.3.9}

$(ch?, m\ \mathrel{:=}_{\mathrm{snc}}\ \mathbf{true}, ch.in_c;\ \mathtt{sync};\ P) \lhd ch!.in_c \rhd (ch?\ \mathrel{:=}_{\mathrm{snc}}\ \mathbf{true};\ \mathtt{sync};\ G^i(n))$

= {Definition of $G^n(\bot)$}

$G^{i+1}(\bot)$

With this result we can show that:

$\mathbf{priAlt}\,\{\mathbf{case}\ ch?x\mathbf{:}\ P\ \mathbin{;}\ \mathbf{break}\}$

= {Definitions 4.1.9 and 4.1.10}

$\mathbf{var\ res}\ \mathrel{:=}_{\mathrm{snc}}\ \mathbf{false}; \mu X \bullet \amalg \lhd \mathbf{res} \rhd$

$\quad (ch?\ \mathrel{:=}_{\mathrm{snc}}\ \mathbf{true}; (\mathbf{res}, m\ \mathrel{:=}_{\mathrm{snc}}\ \mathbf{true}, ch.in_c;\ \mathtt{sync};\ P \lhd ch!.in_c \rhd \amalg)); (\amalg \lhd \mathbf{res} \rhd \mathtt{sync}; X);$

$\mathbf{end\ res}$

= {Laws 3.3.33, predicate calculus}

$\mathbf{var\ res}\ \mathrel{:=}_{\mathrm{snc}}\ \mathbf{false}; \mu X \bullet \amalg \lhd \mathbf{res} \rhd$

$\quad (ch?\ \mathrel{:=}_{\mathrm{snc}}\ \mathbf{true}; (\mathbf{res}, m\ \mathrel{:=}_{\mathrm{snc}}\ \mathbf{true}, ch.in_c;\ \mathtt{sync};\ P \lhd ch!.in_c \rhd \amalg); \amalg \lhd \mathbf{res} \rhd \mathtt{sync}; X);$

$\mathbf{end\ res}$

= {Laws 3.3.33, predicate calculus}

$\mathbf{var\ res}\ \mathrel{:=}_{\mathrm{snc}}\ \mathbf{false}; \mu X \bullet \amalg \lhd \mathbf{res} \rhd$

$\quad (ch?\ \mathrel{:=}_{\mathrm{snc}}\ \mathbf{true}; (\mathbf{res}, m\ \mathrel{:=}_{\mathrm{snc}}\ \mathbf{true}, ch.in_c;\ \mathtt{sync};\ P; \amalg \lhd ch!.in_c \rhd \amalg;\ \mathtt{sync}; X));\ \mathbf{end\ res}$

= {Assumption (P is **SH3**), laws 3.3.10, 3.3.9}

$\mathbf{var\ res}\ \mathrel{:=}_{\mathrm{snc}}\ \mathbf{false}; \mu X \bullet \amalg \lhd \mathbf{res} \rhd$

$\quad (ch?\ \mathrel{:=}_{\mathrm{snc}}\ \mathbf{true}; (\mathbf{res}, m\ \mathrel{:=}_{\mathrm{snc}}\ \mathbf{true}, ch.in_c;\ \mathtt{sync};\ P \lhd ch!.in_c \rhd \mathtt{sync}; X));\ \mathbf{end\ res}$

= {Predicate calculus}

$\mathbf{var\ res}\ \mathrel{:=}_{\mathrm{snc}}\ \mathbf{false}; \mu X \bullet \amalg \lhd \mathbf{res} \rhd$

$\quad (ch?\ \mathrel{:=}_{\mathrm{snc}}\ \mathbf{true}; \mathbf{res}, m\ \mathrel{:=}_{\mathrm{snc}}\ \mathbf{true}, ch.in_c;\ \mathtt{sync};\ P \lhd ch!.in_c \rhd ch?\ \mathrel{:=}_{\mathrm{snc}}\ \mathbf{true};\ \mathtt{sync}; X);\ \mathbf{end\ res}$

= {Definition of F above, laws 2.3.139, 2.3.136 and 2.3.135}

$\bigsqcup\ \mathbf{var\ res}\ \mathrel{:=}_{\mathrm{snc}}\ \mathbf{false}; F(\bot);\ \mathbf{end\ res}$

= {Result above}

$\bigsqcup\ G(\bot)$

= {Law 2.3.139 and definition of G above}

$\mu X \bullet ch?\ \mathrel{:=}_{\mathrm{snc}}\ \mathbf{true}; (m\ \mathrel{:=}_{\mathrm{snc}}\ ch.in_c;\ \mathtt{sync};\ P \lhd ch!.in_c \rhd \mathtt{sync}; X)$

= {Law 3.3.103, predicate calculus, definition 4.1.7}

$ch?x\ \mathbin{;}\ P$ □

Theorem 4.2.2 $\mathbf{delay} = I\!I_1$

Proof.

Straightforward from definitions 4.2.1 and 4.1.5. □

Theorem 4.2.4 $x \overset{:=}{_{\mathsf{HC}}} e = (x := e)_1$

Proof.

Straightforward from definitions 4.2.3 and 4.1.1. □

Law 4.2.5 $(\mathtt{var}\ x; P; (x := e)_1; Q; \mathtt{end}\ x) = (\mathtt{var}\ x; P; (x := e)_1; (x = e)^{\top}_{\mathsf{S}}; Q; \mathtt{end}\ x)$

Proof.

Straightforward from the fact that the law is in the context of the theory synchronous designs (this provides the feedback loop for the variable x) and the fact that the variable is local (it will not be shared by any other program executing in sequence or in parallel with the program where the variable is declared. The result follows directly from law 3.3.164. □

Law 4.2.6 $(\mathtt{var}\ x; P; (x := e)_1; Q; \mathtt{end}\ x) = (\mathtt{var}\ x; P; (x := e)_1; (x = e)^{\bot}_{\mathsf{S}}; Q; \mathtt{end}\ x)$

Proof.

Similar to the proof of law 4.2.5. □

Law 4.2.9 $(b)^{\top}; b * P \blacktriangleright Q = (b)^{\top}; P; b * (P) \blacktriangleright Q$

Proof.

Straightforward from definition 4.2.7 and law 3.3.100. □

Law 4.2.10 $(\neg b)^{\top}; b * P \blacktriangleright Q = (\neg b)^{\top}; Q$

Proof.

Straightforward from definition 4.2.7 and law 3.3.98. □

Law 4.2.17 $ch?x = \mu X \bullet \mathbf{in\text{-}req}(ch); ((x := \mathbf{in}(ch))_1 \lhd \mathbf{wr}(ch) \rhd \mathbf{delay}; X)$

Proof.

Straightforward from definitions 4.1.7, 4.2.11, 4.2.14 and 4.2.15. □

Law 4.2.18 $ch!e = \mu X \bullet \mathbf{out\text{-}req}(ch); \mathbf{out}(ch, e); (\mathbf{delay} \lhd \mathbf{rd}(ch) \rhd \mathbf{delay}; X)$

Proof.

Straightforward from definitions 4.1.8, 4.2.12, 4.2.13 and 4.2.16. □

Law 4.2.19 $(\mathbf{wr}(ch))^{\top}_{\mathsf{S}}; \mathbf{in\text{-}req}(ch) = \mathbf{in\text{-}req}(ch); (\mathbf{wr}(ch))^{\top}_{\mathsf{S}}$

Proof.

Straightforward from definitions 4.2.14 and 4.2.11, and law 3.3.56. □

Law 4.2.20 $(\mathbf{rd}(ch))^{\top}_{\mathsf{S}}; \mathbf{out\text{-}req}(ch) = \mathbf{out\text{-}req}(ch); (\mathbf{rd}(ch))^{\top}_{\mathsf{S}}$

Proof.

Straightforward from definitions 4.2.13 and 4.2.12, and law 3.3.56. □

Law 4.2.21 $(\neg\mathbf{rd}(ch))^{\top}_{S}; \mathbf{out}(ch, e) = (\neg\mathbf{rd}(ch))^{\top}_{S}; \mathbf{out}(ch, e); (\neg\mathbf{rd}(ch))^{\top}_{S}$

Proof.

$$
\begin{aligned}
& (\neg\mathbf{rd}(ch))^{\top}_{S}; \mathbf{out}(ch, e); (\neg\mathbf{rd}(ch))^{\top}_{S} \\
= \; & \{\text{Definitions 3.3.34 and 4.2.16, law 3.3.32}\} \\
& (\neg\mathbf{rd}(ch))^{\top}_{S}; ((ch \mathrel{:=}_{\text{snc}} e; \text{II}) \lhd \neg\mathbf{rd}(ch)[e/ch] \rhd (ch \mathrel{:=}_{\text{snc}} e; \top)) \\
= \; & \{\text{Predicate calculus, } \mathbf{rd}(ch) \text{ does not mention } ch\} \\
& (\neg\mathbf{rd}(ch))^{\top}_{S}; (ch \mathrel{:=}_{\text{snc}} e \lhd \neg\mathbf{rd}(ch) \rhd \top) \\
= \; & \{\text{Definition 3.3.34; laws 3.3.33 and 3.3.7}\} \\
& (ch \mathrel{:=}_{\text{snc}} e \lhd \neg\mathbf{rd}(ch) \rhd \top) \lhd \neg\mathbf{rd}(ch) \rhd \top \\
= \; & \{\text{Laws 3.3.26, 3.3.7 and 3.3.63}\} \\
& (\text{II} \lhd \neg\mathbf{rd}(ch) \rhd \top); ch \mathrel{:=}_{\text{snc}} e \\
= \; & \{\text{Definitions 3.3.34 and 4.2.16}\} \\
& (\neg\mathbf{rd}(ch))^{\top}_{S}; \mathbf{out}(ch, e)
\end{aligned}
$$

□

Law 4.2.22 $(\mathbf{in\text{-}req}(ch); P) \lhd \mathbf{wr}(ch) \rhd (\mathbf{in\text{-}req}(ch); Q) = \mathbf{in\text{-}req}(ch); (P \lhd \mathbf{wr}(ch) \rhd Q)$

Proof.

Straightforward from definitions 4.2.11 and 4.2.14 and law 3.3.32. □

Law 4.2.23 $(\mathbf{out\text{-}req}(ch); P) \lhd \mathbf{rd}(ch) \rhd (\mathbf{out\text{-}req}(ch); Q) = \mathbf{out\text{-}req}(ch); (P \lhd \mathbf{rd}(ch) \rhd Q)$

Proof.

Straightforward from definitions 4.2.12 and 4.2.13 and law 3.3.32. □

Law 4.2.24 $(\mathbf{out}(ch, e); P) \lhd \mathbf{rd}(ch) \rhd (\mathbf{out}(ch, e); Q) = \mathbf{out}(ch, e); (P \lhd \mathbf{rd}(ch) \rhd Q)$

Proof.

Straightforward from definitions 4.2.16 and 4.2.13 and law 3.3.32. □

Law 4.2.25 $\neg\mathbf{DIR\text{-}req}(ch); \mathbf{DIR\text{-}req}(ch) = \mathbf{DIR\text{-}req}(ch)$

Proof.

Direct from definitions 4.2.11 and 4.2.12 and law 3.3.16. □

Law 4.2.26 $\mathbf{DIR\text{-}req}(ch); \neg\mathbf{DIR\text{-}req}(ch) = \neg\mathbf{DIR\text{-}req}(ch)$

Proof.

Similar to proof of law 4.2.25. □

Law 4.2.27 $\mathbf{in\text{-}req}(ch); (v \mathrel{:=}_{\text{snc}} e)_1 = \mathbf{in\text{-}req}(ch)_1 \parallel_{\hat{M}} (v \mathrel{:=}_{\text{snc}} e)_1$

Proof.

$$
\begin{aligned}
&\mathbf{in\text{-}req}(ch); (v \mathrel{:=}_{\mathrm{snc}} e)_1 \\
&= \{\text{Definitions 4.2.3, 4.2.11}\} \\
&\quad ch? \mathrel{:=}_{\mathrm{snc}} \mathbf{true}; v \mathrel{:=}_{\mathrm{snc}} e; \mathsf{sync} \\
&= \{\text{Law 3.3.16, predicate calculus}\} \\
&\quad \mathbf{true} \rightarrow (ch?, v \mathrel{:=}_{\mathrm{snc}} \mathbf{true}, e); \mathsf{sync} \\
&= \{\text{Law 3.3.155}\} \\
&\quad \mathbf{true} \rightarrow (ch? \mathrel{:=}_{\mathrm{snc}} \mathbf{true}); \mathsf{sync} \parallel_{\hat{M}} \mathbf{true} \rightarrow (v \mathrel{:=}_{\mathrm{snc}} e); \mathsf{sync} \\
&= \{\text{Predicate calculus, Definitions 4.2.3, 4.2.11}\} \\
&\quad \mathbf{in\text{-}req}(ch)_1 \parallel_{\hat{M}} (v \mathrel{:=}_{\mathrm{snc}} e)_1
\end{aligned}
$$

□

Law 4.2.28 $\mathbf{out\text{-}req}(ch); (v \mathrel{:=}_{\mathrm{snc}} e)_1 = \mathbf{out\text{-}req}(ch)_1 \parallel_{\hat{M}} (v \mathrel{:=}_{\mathrm{snc}} e)_1$

Proof.

Similar to the proof of law 4.2.27. □

Law 4.2.29 Provided there are no external requests for communication over ch we have that:

$$(s_1 \vee s_2) \overrightarrow{\ }_{\mathrm{snc}} \mathbf{in\text{-}req}(ch)_1 \parallel_{\hat{M}} b \overrightarrow{\ }_{\mathrm{snc}} (v \mathrel{:=}_{\mathrm{snc}} e)_1 = b \overrightarrow{\ }_{\mathrm{snc}} (v \mathrel{:=}_{\mathrm{snc}} e)_1[(s_1 \vee s_2)/\mathbf{rd}(ch)]$$

Proof.

$$
\begin{aligned}
&(s_1 \vee s_2) \overrightarrow{\ }_{\mathrm{snc}} \mathbf{in\text{-}req}(ch)_1 \parallel_{\hat{M}} b \overrightarrow{\ }_{\mathrm{snc}} (v \mathrel{:=}_{\mathrm{snc}} e)_1 \\
&= \{\text{Definitions 3.3.149 and 4.2.11}\} \\
&\quad ((ch?_c \mathrel{:=}_{\mathrm{snc}} \mathbf{true}; \mathsf{sync}) \lhd (s_1 \vee s_2) \rhd \mathsf{sync}) \parallel_{\hat{M}} (v \mathrel{:=}_{\mathrm{snc}} e)_1 \lhd b \rhd \mathsf{sync} \\
&= \{\text{Law 3.3.9, definition 3.3.124, predicate calculus}\} \\
&\quad (ch?_c \mathrel{:=}_{\mathrm{snc}} (s_1 \vee s_2); \mathsf{sync}) \parallel_{\hat{M}} ((v \mathrel{:=}_{\mathrm{snc}} e)_1 \lhd b \rhd \mathsf{sync}) \\
&= \{\text{Laws 3.3.140, 3.3.145 and 3.3.146}\} \\
&\quad ((ch?_c, v \mathrel{:=}_{\mathrm{snc}} (s_1 \vee s_2), e); \mathsf{sync}) \lhd b \rhd (ch?_c \mathrel{:=}_{\mathrm{snc}} (s_1 \vee s_2); \mathsf{sync})) \\
&= \{ch?_c \mathrel{:=}_{\mathrm{snc}} (s_1 \vee s_2) \text{ is executed unconditionally, } b \text{ does not depend on } ch?_c, \text{ definition 3.3.149}\} \\
&\quad ch?_c \mathrel{:=}_{\mathrm{snc}} (s_1 \vee s_2); b \overrightarrow{\ }_{\mathrm{snc}} (v \mathrel{:=}_{\mathrm{snc}} e)_1 \\
&= \{ch?_c \mathrel{:=}_{\mathrm{snc}} (s_1 \vee s_2) \text{ is } \mathbf{S3}\text{, no external requests over } ch\text{, synchronous theory}\} \\
&\quad (ch?.out_c \mathrel{:=}_{\mathrm{snc}} (s_1 \vee s_2) \wedge (ch?.in' = ch?.in); b \overrightarrow{\ }_{\mathrm{snc}} (v \mathrel{:=}_{\mathrm{snc}} e)_1) \wedge (ch?.in \mathrel{=}_{[0\,..\,c]} ch?.out') \\
&= \{\text{Law 3.3.166, predicate calculus, inverse steps}\} \\
&\quad ch?.in_c \mathrel{:=}_{\mathrm{snc}} (s_1 \vee s_2); b \overrightarrow{\ }_{\mathrm{snc}} (v \mathrel{:=}_{\mathrm{snc}} e)_1 \\
&= \{\text{Law 3.3.16 and definition 4.2.13}\} \\
&\quad b \overrightarrow{\ }_{\mathrm{snc}} (v \mathrel{:=}_{\mathrm{snc}} e)_1[(s_1 \vee s_2)/\mathbf{rd}(ch)]
\end{aligned}
$$

□

Law 4.2.30 Provided there are no external requests for communication over ch we have that:

$$(s_1 \vee s_2) \overrightarrow{;}_{\text{snc}} \mathbf{out\text{-}req}(ch)_1 \parallel_{\hat{M}} b \overrightarrow{;}_{\text{snc}} (v :=_{\text{snc}} e)_1 = b \overrightarrow{;}_{\text{snc}} (v :=_{\text{snc}} e)_1[(s_1 \vee s_2)/\mathbf{wr}(ch)]$$

Proof.

Similar to the proof of law 4.2.29. □

Law 4.2.31 Provided that $ch = \texttt{ARB}$, and there are no external requests for communication over ch we have that:

$$s_1 \overrightarrow{;}_{\text{snc}} \mathbf{out}(ch, e_1)_1 \parallel_{\hat{M}} s_2 \overrightarrow{;}_{\text{snc}} \mathbf{out}(ch, e_2)_1 \parallel_{\hat{M}} b \overrightarrow{;}_{\text{snc}} (v :=_{\text{snc}} e)_1 \sqsubseteq$$
$$b \overrightarrow{;}_{\text{snc}} (v :=_{\text{snc}} e)_1[\texttt{SELECT}(\texttt{ARB}, (e_1 \lhd s_1 \rhd \texttt{ARB}), (e_2 \lhd s_2 \rhd \texttt{ARB}))/\mathbf{in}(ch)]$$

Proof.

$$\begin{aligned}
& s_1 \overrightarrow{;}_{\text{snc}} \mathbf{out}(ch, e_1)_1 \parallel_{\hat{M}} s_2 \overrightarrow{;}_{\text{snc}} \mathbf{out}(ch, e_2)_1 \parallel_{\hat{M}} b \overrightarrow{;}_{\text{snc}} (v :=_{\text{snc}} e)_1 \\
= & \ \{\text{Definition 4.2.16}\} \\
& s_1 \overrightarrow{;}_{\text{snc}} (ch :=_{\text{snc}} e_1)_1 \parallel_{\hat{M}} s_2 \overrightarrow{;}_{\text{snc}} (ch :=_{\text{snc}} e_2)_1 \parallel_{\hat{M}} b \overrightarrow{;}_{\text{snc}} (v :=_{\text{snc}} e)_1 \\
= & \ \{\text{Definition 3.3.149, law 3.3.28, assumption } (ch = \texttt{ARB})\} \\
& (ch :=_{\text{snc}} (e_1 \lhd s_1 \rhd \texttt{ARB}))_1 \parallel_{\hat{M}} (ch :=_{\text{snc}} (e_2 \lhd s_2 \rhd \texttt{ARB}))_1 \parallel_{\hat{M}} b \overrightarrow{;}_{\text{snc}} (v :=_{\text{snc}} e)_1 \\
\sqsubseteq & \ \{\text{Similar reasoning to the one applied in proof of law 4.2.29}\} \\
& (ch.out_c :=_{\text{snc}} \texttt{SELECT}(\texttt{ARB}, (e_1 \lhd s_1 \rhd \texttt{ARB}), (e_2 \lhd s_2 \rhd \texttt{ARB})); b \overrightarrow{;}_{\text{snc}} (v :=_{\text{snc}} e)_1 \\
= & \ \{\text{Law 3.3.16 and predicate calculus}\} \\
& b \overrightarrow{;}_{\text{snc}} (v :=_{\text{snc}} e)_1[\texttt{SELECT}(\texttt{ARB}, (e_1 \lhd s_1 \rhd \texttt{ARB}), (e_2 \lhd s_2 \rhd \texttt{ARB}))/ch.out_c] \\
= & \ \{\text{Synchronous theory, } P \text{ is } \mathbf{S2}\text{, predicate calculus}\} \\
& b \overrightarrow{;}_{\text{snc}} (v :=_{\text{snc}} e)_1[\texttt{SELECT}(\texttt{ARB}, (e_1 \lhd s_1 \rhd \texttt{ARB}), (e_2 \lhd s_2 \rhd \texttt{ARB}))/ch.in_c] \\
= & \ \{\text{Definition 4.2.15}\} \\
& b \overrightarrow{;}_{\text{snc}} (v :=_{\text{snc}} e)_1[\texttt{SELECT}(\texttt{ARB}, (e_1 \lhd s_1 \rhd \texttt{ARB}), (e_2 \lhd s_2 \rhd \texttt{ARB}))/\mathbf{in}(ch)]
\end{aligned}$$ □

Law 4.2.32 Provided there are no external **in-req**(ch) events we have that:

$$(v :=_{\text{snc}} e)_1 = (v :=_{\text{snc}} e)_1[\mathbf{false}/\mathbf{rd}(ch)]$$

Proof.

$$\begin{aligned}
& (v :=_{\text{snc}} e)_1 \\
= & \ \{\text{Definitions 3.3.13 and 3.3.124, theorem 3.3.6, predicate calculus}\} \\
& \mathbf{S}(\mathbf{true} \vdash v' = e; \mathbf{sync}) \\
= & \ \{\text{Assumption (no external } \mathbf{in\text{-}req}(ch) \text{ events), 3.3.124 and 4.2.11, law 3.3.16}\} \\
& \mathbf{S}(\mathbf{true} \vdash v' = e; \mathbf{sync} \wedge ch?.out'_c = \mathbf{false})
\end{aligned}$$

= {Synchronous theory ($ch?.in_{[0..c]}^{=} ch?.out$), definition 2.3.1 and predicate calculus}

$(v \overset{:=}{_{\mathsf{snc}}} e; \mathsf{sync})[\mathbf{false}/ch?.in_c]$

= {Definitions 4.2.13 and 4.2.1}

$(v \overset{:=}{_{\mathsf{snc}}} e)_1[\mathbf{false}/\mathbf{rd}(ch)]$ □

Law 4.2.33 Provided there are no external **out-req**(ch) events we have that:

$$(v \overset{:=}{_{\mathsf{snc}}} e)_1 = (v \overset{:=}{_{\mathsf{snc}}} e)_1[\mathbf{false}/\mathbf{wr}(ch)]$$

Proof.

Similar to proof of law 4.2.32. □

Law 4.2.37 $g^\top ; \mathbf{case}\ (a; g\ ?\ P_1 \mid P_2) = g^\top ; a; P_1$

Proof.

Direct from definitions 4.2.36 and 4.2.34 together with law 3.3.58. □

Law 4.2.38 $(\neg g)^\top ; \mathbf{case}\ (a; g\ ?\ P_1 \mid P_2) = (\neg g)^\top ; a; P_2$

Proof.

Direct from definitions 4.2.36 and 4.2.34 together with law 3.3.59. □

Law 4.2.39 $(g_1)_{\mathsf{S}}^\top ; \mathbf{case}\ (a_1; g_1\ ?\ P_1 \mid \cdots \mid a_n; g_n\ ?\ P_n \mid P) = (g_1)_{\mathsf{S}}^\top ; a_1; P_1$

Proof.

Direct from definitions 4.2.36 and 4.2.34 together with law 3.3.58. □

Law 4.2.40 $(\neg g_1)_{\mathsf{S}}^\top ; \mathbf{case}\ (a_1; g_1\ ?\ P_1 \mid a_2; g_2\ ?\ P_2 \mid \ldots) = (\neg g_1)_{\mathsf{S}}^\top ; a_1; \mathbf{case}\ (a_2; g_2\ ?\ P_2 \mid \ldots)$

Proof.

Direct from definitions 4.2.36 and 4.2.34 together with law 3.3.59. □

Law 4.2.43 $(chk(g))^\top ; req(g) = req(g); (chk(g))^\top$

Proof.

By case analysis on g together with laws 4.2.19 and 4.2.20. □

Law 4.2.44 $(req(g); P) \lhd chk(g) \rhd (req(g); Q) = req(g); (P \lhd chk(g) \rhd Q)$

Proof.

Direct by case analysis on g, definitions 4.2.42 and 4.2.41 and laws 4.2.22 , 4.2.23 and 4.2.24.

□

Law 4.2.46 Prialt with default clause equivalence

$$\mathbf{priAlt}\left\{\begin{array}{l}\mathbf{case}\ g_1\mathbf{:}\ P_1 \fatsemi \mathbf{break}\fatsemi \\ \vdots \\ \mathbf{case}\ g_n\mathbf{:}\ P_n \fatsemi \mathbf{break}\fatsemi \\ \mathbf{default:}\ P_d\end{array}\right\} = \mathbf{case}\left(\begin{array}{l} req(g_1); chk(g_1)\ ?\ act(g_1); P_1 \mid \\ \vdots \\ req(g_n); chk(g_n)\ ?\ act(g_n); P_n \mid \\ P_d\end{array}\right)$$

Proof.

From law 4.1.24 it is enough to show the validity of this law for the case of a priAlt with one case expression and a **default** statement. To keep the proof concise, we will assume the first guard performs an input over channel ch_1 (a similar proof will also hold for an output guard).

$$\begin{array}{l}\quad \mathbf{priAlt}\,\{\mathbf{case}\ ch_1?x\mathbf{:}\ P_1 \fatsemi \mathbf{break} \fatsemi \mathbf{default:}\ P_2\} \\ = \{\text{Lemma C.0.22}\} \\ \quad ch_1? \mathrel{\overset{:=}{\scriptstyle \mathrm{snc}}} \mathbf{true}; (m \mathrel{\overset{:=}{\scriptstyle \mathrm{snc}}} ch_1.in_c; \mathbf{sync}; P_1 \lhd ch_1!.in_c \rhd P_2) \\ = \{\text{Definitions 4.2.11, 4.2.14, 4.2.15, 4.2.3 and 4.2.34}\} \\ \quad \mathbf{case}\ (\mathbf{in\text{-}req}(ch_1); \mathbf{wr}(ch_1); (m \mathrel{\overset{:=}{\scriptstyle \mathrm{snc}}} \mathbf{in}(ch))_1\ ?\ P_2)\end{array}$$

□

Law 4.2.47 Prialt without default clause equivalence

$$\mathbf{priAlt}\left\{\begin{array}{l}\mathbf{case}\ g_1\mathbf{:}\ P_1 \fatsemi \mathbf{break};\\ \vdots \\ \mathbf{case}\ g_n\mathbf{:}\ P_n \fatsemi \mathbf{break}\end{array}\right\} = \mu X \bullet \mathbf{case}\left(\begin{array}{l} req(g_1); chk(g_1)\ ?\ act(g_1); P_1 \mid \\ \vdots \\ req(g_n); chk(g_n)\ ?\ act(g_n); P_n \mid \\ \mathit{Ⅱ}_1; X\end{array}\right)$$

Proof.

Similar to the proof of law 4.1.25. □

Appendix D

Proofs from Chapter 5

Law 5.1.9 $*(b \overrightarrow{\ }_{\text{snc}} P) = (P \lhd b \rhd \text{II}); *(b \overrightarrow{\ }_{\text{snc}} P)$

Proof.

$$
\begin{aligned}
& * (b \overrightarrow{\ }_{\text{snc}} P) \\
= & \ \{\text{Definitions 5.1.8 and 3.3.96}\} \\
& \mu X \bullet (b \overrightarrow{\ }_{\text{snc}} P; X) \lhd b \rhd \text{II} \\
= & \ \{\text{Law 3.3.57, predicate calculus and definitions 5.1.8 and 3.3.96}\} \\
& (b^{\top}_{\text{S}}; P; *(b \overrightarrow{\ }_{\text{snc}} P)) \lhd b \rhd (\neg b)^{\top}_{\text{S}} \\
= & \ \{\text{Law 3.3.98}\} \\
& (b^{\top}_{\text{S}}; P; *(b \overrightarrow{\ }_{\text{snc}} P)) \lhd b \rhd ((\neg b)^{\top}_{\text{S}}; *(b \overrightarrow{\ }_{\text{snc}} P)) \\
= & \ \{\text{Laws 3.3.57 and 3.3.33}\} \\
& (P \lhd b \rhd \text{II}); *(b \overrightarrow{\ }_{\text{snc}} P)
\end{aligned}
$$

□

Law 5.1.10 Provided $\neg(b \wedge c)$ then we have:

$$b^{\top}; *((b \overrightarrow{\ }_{\text{snc}} P) \parallel_{\hat{M}} (c \overrightarrow{\ }_{\text{snc}} Q)) = b^{\top}; P; *((b \overrightarrow{\ }_{\text{snc}} P) \parallel_{\hat{M}} (c \overrightarrow{\ }_{\text{snc}} Q))$$

Proof.

$$
\begin{aligned}
& b^{\top}; *((b \overrightarrow{\ }_{\text{snc}} P) \parallel_{\hat{M}} (c \overrightarrow{\ }_{\text{snc}} Q)) \\
= & \ \{\text{Laws 3.3.153, 5.1.9}\} \\
& b^{\top}; (((b \overrightarrow{\ }_{\text{snc}} P) \parallel_{\hat{M}} (c \overrightarrow{\ }_{\text{snc}} Q)) \lhd b \vee c \rhd \text{II})); *((b \overrightarrow{\ }_{\text{snc}} P) \parallel_{\hat{M}} (c \overrightarrow{\ }_{\text{snc}} Q)) \\
= & \ \{\text{Laws 3.3.58 and 3.3.154}\} \\
& b^{\top}; b \overrightarrow{\ }_{\text{snc}} P; *((b \overrightarrow{\ }_{\text{snc}} P) \parallel_{\hat{M}} (c \overrightarrow{\ }_{\text{snc}} Q)) \\
= & \ \{\text{Law 3.3.150}\} \\
& b^{\top}; P; *((b \overrightarrow{\ }_{\text{snc}} P) \parallel_{\hat{M}} (c \overrightarrow{\ }_{\text{snc}} Q))
\end{aligned}
$$

□

Lemma D.0.23. $*(b \overset{\rightarrow}{_{snc}} P) = b * (b^{\top}_{S}; b \overset{\rightarrow}{_{snc}} P)$

Proof.

$$
\begin{aligned}
& * (b \overset{\rightarrow}{_{snc}} P) \\
= & \{\text{Definitions 5.1.8 and 3.3.96}\} \\
& \mu X \bullet (b \overset{\rightarrow}{_{snc}} P; X) \lhd b \rhd \text{II} \\
= & \{\text{Laws 3.3.57 and 3.3.50}\} \\
& \mu X \bullet (b^{\top}_{S}; b^{\top}_{S}; b \overset{\rightarrow}{_{snc}} P; X) \lhd b \rhd ((\neg b)^{\top}_{S}; \text{II}) \\
= & \{\text{Laws 3.3.57}\} \\
& \mu X \bullet (b^{\top}_{S}; b \overset{\rightarrow}{_{snc}} P; X) \lhd b \rhd \text{II} \\
= & \{\text{Definition 3.3.96}\} \\
& b * (b^{\top}_{S}; b \overset{\rightarrow}{_{snc}} P)
\end{aligned}
$$

□

Law 5.1.11 Provided $\neg(b \wedge c)$ and P takes at least one clock cycle, then we have:

$$*(b \overset{\rightarrow}{_{snc}} P) = b * ((b \overset{\rightarrow}{_{snc}} P) \parallel_{\hat{M}} (c \overset{\rightarrow}{_{snc}} Q))$$

Proof.

$$
\begin{aligned}
& * (b \overset{\rightarrow}{_{snc}} P) \\
= & \{\text{Lemma D.0.23}\} \\
& b * (b^{\top}_{S}; b \overset{\rightarrow}{_{snc}} P) \\
= & \{\text{Assumption } (\neg(b \wedge c)), \text{ propositional calculus}\} \\
& b * ((b \wedge \neg c)^{\top}_{S}; b \overset{\rightarrow}{_{snc}} P) \\
= & \{\text{Law 3.3.154}\} \\
& b * ((b \wedge \neg c)^{\top}_{S}; (b \overset{\rightarrow}{_{snc}} P \parallel_{\hat{M}} c \overset{\rightarrow}{_{snc}} Q) \\
= & \{\text{Inverse steps}\} \\
& b * ((b \overset{\rightarrow}{_{snc}} P) \parallel_{\hat{M}} (c \overset{\rightarrow}{_{snc}} Q))
\end{aligned}
$$

□

Law 5.1.12 Provided $inh\,(b \overset{\rightarrow}{_{snc}} P, b \overset{\rightarrow}{_{snc}} P)$ we have:

$$*(b \overset{\rightarrow}{_{snc}} P) = P \lhd b \rhd \text{II}$$

Proof.

$$
\begin{aligned}
& * (b \overset{\rightarrow}{_{snc}} P) \\
= & \{\text{Law 5.1.9}\} \\
& (b \overset{\rightarrow}{_{snc}} P \lhd b \rhd \text{II}); b * (b \overset{\rightarrow}{_{snc}} P) \\
= & \{\text{Assumption } (b \overset{\rightarrow}{_{snc}} P \text{ is self-inhibiting}), \text{ laws 5.1.7 and 3.3.57}\}
\end{aligned}
$$

$$
\begin{array}{ll}
 & ((b^{\top}_{\mathrm{S}}; b \rightarrow_{\mathrm{snc}} P; (\neg b)^{\perp}_{\mathrm{S}}) \lhd b \rhd (\neg b)^{\perp}_{\mathrm{S}}); b * (b \rightarrow_{\mathrm{snc}} P) \\
= & \{\text{Laws 3.3.33, 3.3.150 and 3.3.99}\} \\
 & b^{\top}_{\mathrm{S}}; P; (\neg b)^{\perp}_{\mathrm{S}} \lhd b \rhd (\neg b)^{\perp}_{\mathrm{S}} \\
= & \{\text{Laws 5.1.7 and 3.3.57 }\} \\
 & P \lhd b \rhd \mathrm{II} \qquad \square
\end{array}
$$

Law 5.1.13 Provided $inh\,(P, Q)$ then

$$(c \notin act\,(Q))_{\perp}; *P = (c \notin act\,(Q))_{\perp}; *P; (c \notin (act\,(Q) \cup act\,(P))_{\perp}$$

Proof.

$$
\begin{array}{ll}
 & (c \notin act\,(Q))_{\perp}; *P \\
= & \{\text{Law 3.3.102}\} \\
 & (c \notin act\,(Q))_{\perp}; *P; (c \notin act\,(P))_{\perp} \\
= & \{\text{Assumption } (inh\,(P, Q)), \text{ laws 5.1.7 and 3.3.102}\} \\
 & (c \notin act\,(Q))_{\perp}; *P; (c \notin act\,(Q))_{\perp}; (c \notin act\,(P))_{\perp} \\
= & \{\text{Law 3.3.47 and propositional calculus}\} \\
 & (c \notin act\,(Q))_{\perp}; *P; (c \notin (act\,(Q) \cup act\,(P))_{\perp} \qquad \square
\end{array}
$$

Law 5.1.14 Provided $\neg(b \wedge c)$ and $inh\,(c \rightarrow_{\mathrm{snc}} Q, b \rightarrow_{\mathrm{snc}} P)$ then we have

$$*((b \rightarrow_{\mathrm{snc}} P) \parallel_{\hat{M}} (c \rightarrow_{\mathrm{snc}} Q)) = *(b \rightarrow_{\mathrm{snc}} P); *(c \rightarrow_{\mathrm{snc}} Q)$$

Proof.

$$
\begin{array}{ll}
 & *\,((b \rightarrow_{\mathrm{snc}} P) \parallel_{\hat{M}} (c \rightarrow_{\mathrm{snc}} Q)) \\
= & \{\text{Law 3.3.104 (twice)}\} \\
 & (b * ((b \rightarrow_{\mathrm{snc}} P) \parallel_{\hat{M}} (c \rightarrow_{\mathrm{snc}} Q))); (c * ((b \rightarrow_{\mathrm{snc}} P) \parallel_{\hat{M}} (c \rightarrow_{\mathrm{snc}} Q))); *((b \rightarrow_{\mathrm{snc}} P) \parallel_{\hat{M}} (c \rightarrow_{\mathrm{snc}} Q)) \\
= & \{\text{Assumption } (\neg(b \wedge c)), \text{ law 5.1.11 (twice)}\} \\
 & *\,(b \rightarrow_{\mathrm{snc}} P); *(c \rightarrow_{\mathrm{snc}} Q); *((b \rightarrow_{\mathrm{snc}} P) \parallel_{\hat{M}} (c \rightarrow_{\mathrm{snc}} Q)) \\
= & \{\text{Laws 3.3.101, 5.1.13}\} \\
 & *\,(b \rightarrow_{\mathrm{snc}} P); *(c \rightarrow_{\mathrm{snc}} Q); (\neg b \wedge \neg c)_{\perp}; *((b \rightarrow_{\mathrm{snc}} P) \parallel_{\hat{M}} (c \rightarrow_{\mathrm{snc}} Q)) \\
= & \{\text{Laws 3.3.99, 3.3.101 and 5.1.13}\} \\
 & *\,(b \rightarrow_{\mathrm{snc}} P); *(c \rightarrow_{\mathrm{snc}} Q) \qquad \square
\end{array}
$$

Law 5.1.15 Provided $\neg(b \wedge c)$, $inh\,(b \rightarrow_{\mathrm{snc}} P, b \rightarrow_{\mathrm{snc}} P)$ and $inh\,(c \rightarrow_{\mathrm{snc}} Q, b \rightarrow_{\mathrm{snc}} P)$ then we have:

$$b^{\top}; *((b \rightarrow_{\mathrm{snc}} P) \parallel_{\hat{M}} (c \rightarrow_{\mathrm{snc}} Q)) = b^{\top}; P; *(c \rightarrow_{\mathrm{snc}} Q)$$

Proof.

$$
\begin{aligned}
& b^{\top}; *((b \overrightarrow{\ }_{\text{snc}} P) \parallel_{\hat{M}} (c \overrightarrow{\ }_{\text{snc}} Q)) \\
= & \{\text{Law 5.1.14 (assumption matches the law's assumptions)}\} \\
& b^{\top}; *(b \overrightarrow{\ }_{\text{snc}} P); *(c \overrightarrow{\ }_{\text{snc}} Q) \\
= & \{\text{Assumption } (inh\,(b \overrightarrow{\ }_{\text{snc}} P, b \overrightarrow{\ }_{\text{snc}} P)), \text{ law 5.1.12}\} \\
& b^{\top}; b \overrightarrow{\ }_{\text{snc}} P; *(c \overrightarrow{\ }_{\text{snc}} Q) \\
= & \{\text{Law 3.3.150}\} \\
& b^{\top}; P; *(c \overrightarrow{\ }_{\text{snc}} Q)
\end{aligned}
$$

□

Law 5.1.16 Provided P and Q take a single clock cycle we have: $*((b \overrightarrow{\ }_{\text{snc}} P) \parallel_{\hat{M}} (b \overrightarrow{\ }_{\text{snc}} Q)) = *(b \overrightarrow{\ }_{\text{snc}} P) \parallel_{\hat{M}} *(b \overrightarrow{\ }_{\text{snc}} Q)$

Proof.

The proof follows a similar proof from [Hoare and He 1998, page 139]. Let:

$$
\begin{aligned}
F(X) &=_{df} (b \overrightarrow{\ }_{\text{snc}} P; X) \lhd b \rhd \text{II} \\
G(X) &=_{df} (b \overrightarrow{\ }_{\text{snc}} P; X) \lhd b \rhd \text{II} \\
S(X) &=_{df} ((b \overrightarrow{\ }_{\text{snc}} P) \parallel_{\hat{M}} (b \overrightarrow{\ }_{\text{snc}} Q)\,; X) \lhd b \rhd \text{II}
\end{aligned}
$$

We want to show that $F^n(\bot) \parallel_{\hat{M}} G^n(\bot) = S^n(\bot)$ by induction on n:
Base case ($n = 0$):

$$
\begin{aligned}
& F^n(\bot) \parallel_{\hat{M}} G^n(\bot) \\
= & \{F^0(\bot) = \bot \text{ for any } F\} \\
& \bot \parallel_{\hat{M}} \bot \\
= & \{\text{Law 3.3.139}\} \\
& \bot \\
= & \{F^0(\bot) = \bot \text{ for any } F\} \\
& S^0(\bot)
\end{aligned}
$$

Inductive case ($n = i + 1$):

$$
\begin{aligned}
& F^{i+1}(\bot) \parallel_{\hat{M}} G^{i+1}(\bot) \\
= & \{\text{Definition of } F \text{ and } G\} \\
& \left((b \overrightarrow{\ }_{\text{snc}} P; F^i(\bot)) \lhd b \rhd \text{II}\right) \parallel_{\hat{M}} \left((b \overrightarrow{\ }_{\text{snc}} Q; G^i(\bot)) \lhd b \rhd \text{II}\right)
\end{aligned}
$$

$$
\begin{aligned}
&= \{\text{Laws 3.3.57, 3.3.150 and 3.3.140 (twice)}\} \\
&\quad \Big(\big((P; F^i(\bot)) \parallel_{\hat{M}} (Q; G^i(\bot))\big) \lhd b \rhd \big((P; F^i(\bot)) \parallel_{\hat{M}} \mathtt{II}\big)\Big) \lhd b \rhd \Big(\big((Q; G^i(\bot)) \lhd b \rhd \mathtt{II}\big)\Big) \\
&= \{\text{Laws 3.3.26, 3.3.142 } (P \text{ and } Q \text{ take exactly one clock cycle})\} \\
&\quad \big((P \parallel_{\hat{M}} Q); (F^i(\bot) \parallel_{\hat{M}} G^i(\bot))\big) \lhd b \rhd \mathtt{II} \\
&= \{\text{Inductive hypothesis, definition of } S\} \\
&\quad S^{i+1}(\bot)
\end{aligned}
$$

With this result we can now prove that:

$$
\begin{aligned}
&\quad * (b \xrightarrow[\text{snc}]{} P) \parallel_{\hat{M}} *(b \xrightarrow[\text{snc}]{} Q) \\
&= \{\text{Definitions of } F \text{ and } G, \text{ law 2.3.139}\} \\
&\quad \left(\bigsqcup_i F^i(\bot)\right) \parallel_{\hat{M}} \left(\bigsqcup_j G^j(\bot)\right) \\
&= \{\text{Law 3.3.141 (twice), diagonalisation}\} \\
&\quad \bigsqcup_i \big(F^i(\bot) \parallel_{\hat{M}} G^i(\bot)\big) \\
&= \{\text{Observation above}\} \\
&\quad \bigsqcup_i \big(S^i(\bot)\big) \\
&= \{\text{Definition of } S \text{ and theorem 2.3.139}\} \\
&\quad * ((b \xrightarrow[\text{snc}]{} P) \parallel_{\hat{M}} (b \xrightarrow[\text{snc}]{} Q))
\end{aligned}
$$

□

Bibliography

Abdel-hamid, A. T., Zaki, M. and Tahar, S. [2004], A tool converting finite state machine to VHDL, *in* 'Proceedings of IEEE Canadian Conference on Electrical & Computer Engineering (CCECE'04), Niagara Falls', pp. 1907–1910.

Aho, A. V., Sethi, R. and Ullman, J. D. [1986], *Compilers: principles, techniques, and tools*, Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA.

Aho, A. V. and Ullman, J. D. [1972], *The theory of parsing, translation, and compiling*, Prentice-Hall, Inc., NJ, USA.

Artan, R. [2000], *ProofPower HOL Reference Manual*, Lemma 1 Ltd. File USR029.DOC.

Ashenden, P. J. [1999], *The Designer's Guide to VHDL*, Morgan Kaufmann Publishers Inc., San Francisco, CA, USA.

Aubury, M., Page, I., Randall, G., Saul, J. and Watts, R. [1996], Handel-C language examples, Technical report, Oxford University Computing Laboratory.

Barrett, G. [1992], *Occam 3 reference manual*, Inmos Ltd.

Barron, I. M. [1978], The transputer, *in* D. Aspinall, ed., 'Microprocessor and Its Application', Cambridge University Press, New York, NY, USA.

Barros, E. and Sampaio, A. [1994], Towards provably correct hardware/software partitioning using Occam, *in* 'International workshop on Hardware/software co-design', IEEE Computer Society Press, Los Alamitos, CA, USA, pp. 210–217.

Beizer, B. [1990], *Software testing techniques (2nd ed.)*, Van Nostrand Reinhold Co., New York, NY, USA.

Berghofer, S. and Strecker, M. [2003], 'Extracting a formally verified, fully executable compiler from a proof assistant', *Electronic Notes in Theoretical Computer Science* **82**(2).

Bergstra, J. A. and Klop, J. W. [1985], 'Algebra of communicating processes with abstraction', *Theoretical Computer Science* **37**, 77–121.

Borba, P., Sampaio, A., Cavalcanti, A. and Cornelio, M. [2004], 'Algebraic reasoning for object-oriented programming', *Science of Computing Programming* **52**(1-3), 53–100.

Boulton, R., Gordon, A., Gordon, M., Harrison, J., Herbert, J. and Tassel, J. V. [1993], Experience with embedding hardware description languages in HOL, *in* V. Stavridou, T. F. Melham and R. T. Boute, eds, 'Proceedings of the International Conference on Theorem Provers Circuit Design: Theory, Practice and Experience', Vol. A-10 of *IFIP Transactions A: Computer Science and Technology*, North-Holland, Nijmegen, The Netherlands, pp. 129–156.

Bowen, J. [1993], 'From Programs to Object Code and back again using Logic Programming: Compilation and Decompilation', *Journal of Software Maintenance: Research and Practice* **5**(4), 205–234.

Bowen, J. and He, J. [2001], 'An approach to specification and verification of a hardware compilation scheme', *The Journal of supercomputing* **19**(1), 23–39.

Bowen, J., He, J. and Page, I. [1994], Hardware compilation, Elsevier, pp. 193–207.

Breuer, P. T., Fernández, L. S. and Kloos, C. D. [1994*a*], Clean formal semantics for VHDL, *in* 'Proceeding of the European Design and Test Conference', IEEE Computer Society Press, Paris, France, pp. 641–647.

Breuer, P. T., Fernández, L. S. and Kloos, C. D. [1994*b*], 'Proof theory and a validation condition generator for VHDL', *Euro-VHDL '94* pp. 512–517.

Breuer, P. T., Fernández, L. S. and Kloos, C. D. [1995], 'A simple denotational semantics, proof theory and a validation condition generator for unit-delay VHDL', *Formal Methods in System Design* **7**(1-2), 27–51.

Burstall, R. M. and Landin, P. J. [1969], Programs and their proofs: An algebraic approach, *in* B. Meltzer and D. Michie, eds, 'Machine Intelligence 4', American Elsevier, New York, pp. 17–44.

Butler, R. W. and Finelli, G. B. [1991], The infeasibility of experimental quantification of life-critical software reliability, *in* 'Proceedings of the conference on Software for critical systems 1993', ACM Press, New York, NY, USA, pp. 66–76.

Butterfield, A. [2001], Interpretative semantics for PriAlt-free Handel-C, Technical report, The University of Dublin, Trinity College.

Butterfield, A. [2007], A denotational semantics for Handel-C, *in* C. B. Jones, Z. Liu and J. Woodcock, eds, 'Formal Methods and Hybrid Real-Time Systems', Vol. 4700 of *Lecture Notes in Computer Science*, Springer, pp. 45–66.

Butterfield, A., Sherif, A. and Woodcock, J. [2007], Slotted-*Circus*, *in* J. Davies and J. Gibbons, eds, 'Integrated Formal Methods 2007', Vol. 4591 of *Lecture Notes in Computer Science*, Springer, pp. 75–97.

Butterfield, A. and Woodcock, J. [2002], 'Semantic domains for Handel-C', *Electronic Notes in Theoretical Computer Science* **74**.

Butterfield, A. and Woodcock, J. [2005*a*], Denotational semantics of Handel-C cores. (Unpublished work).

Butterfield, A. and Woodcock, J. [2005*b*], 'priAlt in Handel-C: an operational semantics', *International Journal on Software Tools Technology Transfer* **7**(3), 248–267.

Butterfield, A. and Woodcock, J. [2006], A Hardware Compiler Semantics for Handel-C, *in* 'Mathematical Foundations of Computer Science and Information Technology 2004', number 161 *in* 'Electronic Notes in Theoretical Computer Science', Dublin, Ireland, pp. 73–90.

Cavalcanti, A. and Naumann, D. [2000], 'A Weakest Precondition Semantics for Refinement of Object-oriented Programs', *IEEE Transactions on Software Engineering* **26**(8), 713–728.
URL: *http://www.cs.kent.ac.uk/pubs/2000/1466*

Cavalcanti, A. and Woodcock, J. [2006], A Tutorial Introduction to CSP in Unifying Theories of Programming, *in* 'Refinement Techniques in Software Engineering', Vol. 3167 of *Lecture Notes in Computer Science*, Springer-Verlag, pp. 220 – 268.

Celoxica Ltd. [2002*a*], *DK3: Handel-C Language Reference Manual*.

Celoxica Ltd. [2002*b*], *The Technology behind DK1*. Application Note AN 18.

Claessen, K. and Pace, G. [2002], An embedded language framework for hardware compilation, *in* 'Designing Correct Circuits', Design Correct Circuits (DCC), ETAPS 2002.
URL: *http://www.cs.chalmers.se/ koen/Papers/dcc-hwcomp.ps*

Cohn, P. [1981], *Universal Algebra*, number 6 *in* 'Mathematics and its Applications', Reidel. Originally published by Harper and Row, 1965.

Colby, C., Lee, P., Necula, G. C., Blau, F., Plesko, M. and Cline, K. [2000], A certifying compiler for Java, *in* 'Proceedings of the ACM SIGPLAN 2000 conference on Programming language design and implementation', ACM, New York, NY, USA, pp. 95–107.

Davey, B. and Priestley, H. [2002], *Introduction to Lattices and Order*, Cambridge University Press.

Davie, J. T. and Morrison, R. [1982], *Recursive Descent Compiling*, John Wiley & Sons.

Davis, J. F. [2005], 'The affordable application of formal methods to software engineering', *Ada Lett.* **XXV**(4), 57–62.

De Nicola, R. and Pugliese, R. [1994], Testing linda: Observational semantics for an asynchronous language, Rapporto di Ricerca SI/RR-94/06, Dipartimento di Scienze dell'Informazione, Università di Roma "La Sapienza".

Demoen, B. [2005], 'Programming in Prolog. using the ISO standard', *Theory and Practice in Logic Programming* **5**(3), 391–395.

Dijkstra, E. W. [1975], 'Guarded commands, nondeterminacy and formal derivation of programs', *Communications of the ACM* **18**(8), 453–457.

Dijkstra, E. W. [1976], *A Discipline of Programming*, Series in Automatic Computation, Prentice Hall.

Dold, A. and Vialard, V. [2001], A mechanically verified compiling specification for a Lisp compiler, *in* 'Conference on Foundations of Software Technology and Theoretical Computer Science 2001', Springer-Verlag, London, UK, pp. 144–155.

Duran, A., Cavalcanti, A. and Sampaio, A. [2003*a*], A refinement strategy for the compilation of classes, inheritance and dynamic binding (extended version), Technical report, Computing laboratory, University of Kent at Canterbury.

Duran, A., Cavalcanti, A. and Sampaio, A. [2003*b*], A strategy for compiling classes, inheritance, and dynamic binding, *in* 'Formal Methods Europe 2003', Springer-Verlag, Pisa, Italy, pp. 301–320.

Duran, A., Sampaio, A. and Cavalcanti, A. [2001], 'Formal bytecode generation for ROOL virtual machine', *IV WMF- Workshop on Formal Methods* .

Erne, M., Koslowski, J., Melton, A. and Strecker, G. E. [1992], A primer on galois connections, *in* 'York Academy of Science'.

Gelperin, D. and Hetzel, B. [1988], 'The growth of software testing', *Communications of the ACM* **31**(6), 687–695.

Glesner, S., Geiß, R. and Boesler, B. [2002], 'Verified code generation for embedded systems', *Electronic Notes in Theoretical Computer Science* **65**(2).

Goerigk, W. [2002], 'Towards acceptability of optimizations: An extended view of compiler correctness', *Electronic Notes in Theoretical Computer Science* **65**(2).

Goerigk, W., Dold, A., Gaul, T., Goos, G., Heberle, A., von Henke, F., Hoffmann, U., Langmaack, H., Pfeifer, H., Ruess, H. and Zimmermann, W. [1996], 'Compiler correctness and implementation verification: The verifix approach'. Compiler Correctness and Implementation Verification: The Verifix Approach. In CC '96 International Conf. on Compiler Construction 1996 (poster session).

Goerigk, W. and Simon, F. H. [1999], Towards rigorous compiler implementation verification, *in* 'Collaboration between Human and Artificial Societies', pp. 62–73.

Goguen, J. A., Thatcher, J. W., Wagner, E. G. and Wright, J. B. [1977], 'Initial algebra semantics and continuous algebras', *Journal of the ACM* **24**(1), 68–95.

Goguen, J. and Winkler, T. [1988], Introducing OBJ3, Technical Report SRI-CSL-88-9, SRI International, Menlo Park.

Goos, G. [2002], 'Compiler verification and compiler architecture', *Electronic Notes in Theoretical Computer Science* **65**(2).

Goossens, K. G. W. [1995], Reasoning about VHDL using operational and observational semantics, *in* P. E. Camurati and H. Eveking, eds, 'Correct Hardware Design Methodologies', Vol. 987, Springer Verlag, pp. 311–327.

Gordon, M. and Melham, T., eds [1993], *Introduction to HOL: a theorem proving environment for higher order logic*, Cambridge University Press.

Grotker, T. [2002], *System Design with SystemC*, Kluwer Academic Publishers, Norwell, MA, USA.

Hartenstein, R. [1997], The microprocessor is no more general purpose: why future reconfigurable platforms will win, *in* 'Proceedings of the Second Annual IEEE International Conference on Innovative Systems in Silicon'.

Harwood, W., Cavalcanti, A. L. C. and Woodcock, J. C. P. [2008], A Theory of Pointers for the UTP, *in* J. S. Fitzgerald, A. E. Haxthausen and H. Yenigun, eds, 'Theoretical Aspects of Computing', Vol. 5160 of *Lecture Notes in Computer Science*, Springer-Verlag, pp. 141 – 155.

He, J. [2002], An algebraic approach to the Verilog programming., *in* '10th Anniversary Colloquium of UNU/IIST', pp. 65–80.

He, J., Bowen, J., and Page, I. [1992], A provably correct hardware implementation of Occam, Technical report, ProCoS II - Oxford University.

He, J., Page, I. and Bowen, J. [1993], Towards a provably correct hardware implementation of Occam, *in* L. Pierre, ed., 'Correct Hardware Design and Verification Methods', Springer-Verlag, pp. 214–225.

Hierons, R. M., Bogdanov, K., Bowen, J. P., Cleaveland, R., Derrick, J., Dick, J., Gheorghe, M., Harman, M., Kapoor, K., Krause, P., Lüttgen, G., Simons, A. J. H., Vilkomir, S., Woodward, M. R. and Zedan, H. [2009], 'Using formal specifications to support testing', *ACM Comput. Surv.* **41**(2), 1–76.

Hoare, C. [1990], Refinement algebra proves correctness of compiling specifications, *in* J. Woodcock, ed., '3rd Refinement Workshop', Springer-Verlag.

Hoare, C. A. R. [1983], 'Communicating Sequential Processes', *Communications of the ACM* **26**(1), 100–106.

Hoare, C. A. R. [1985], Programs are predicates, *in* 'Proceedings of a discussion meeting of the Royal Society of London on Mathematical logic and programming languages', Prentice-Hall, Inc., NJ, USA, pp. 141–155.

Hoare, C. A. R., He, J. and Sampaio, A. [1993], 'Normal form approach to compiler design', *Acta informatica* **30**(9), 701–739.

Hoare, C. A. R., He, J. and Sampaio, A. [2000], *Algebraic derivation of an operational semantics*, Proof, language, and interaction: essays in honour of Robin Milner, MIT Press, pp. 77–98.

Hoare, C. and He, J. [1998], *Unifying Theories of Programming*, Prentice Hall.

Hoffmann, U. [1998], Compiler Implementation Verification through Rigorous Syntactical Code Inspection, PhD thesis, Technical Faculty, CAU Kiel.

Hudak, P., Peyton Jones, S., Wadler, P., Boutel, B., Fairbairn, J., Fasel, J., Guzmán, M. M., Hammond, K., Hughes, J., Johnsson, T., Kieburtz, D., Nikhil, R., Partain, W. and Peterson, J. [1992], 'Report on the programming language haskell: a non-strict, purely functional language version 1.2', *SIGPLAN Not.* **27**(5), 1–164.

IEEE [1993], 'Multivalue Logic System for VHDL Model Interoperability (Std_logic_1164)', IEEE Standard 1164-1993.

ISO [1999], ISO C Standard 1999, Technical report. ISO/IEC 9899:1999 draft.
URL: *http://www.open-std.org/jtc1/sc22/wg14/www/docs/n1124.pdf*

Iyoda, J. [2007], Translating HOL functions to hardware, Technical Report UCAM-CL-TR-682, University of Cambridge, Computer Laboratory.
URL: *http://www.cl.cam.ac.uk/techreports/UCAM-CL-TR-682.pdf*

Iyoda, J. and He, J. [2001*a*], A Prolog prototype for the synthesis of Verilog, Technical Report 237, International Institute for Software Technology, United Nations University.

Iyoda, J. and He, J. [2001*b*], Towards and algebraic synthesis of Verilog, Technical Report 218, International Institute for Software Technology, United Nations University.

Joost, R. and Salomon, R. [2005], Advantages of FPGA-based multiprocessor systems in industrial applications, *in* '31st Annual Conference of IEEE (Industrial Electronics Society)', pp. 6 pp.–.

Kamat, R. K., Shinde, S. A. and Shelake, V. G. [2009], *Unleash the System On Chip using FPGAs and Handel C*, Springer Publishing Company, Incorporated.

Kernighan, B. W. and Ritchie, D. M. [1988], *The C Programming Language*, Prentice Hall Professional Technical Reference.

Klein, G. and Nipkow, T. [2003], 'Verified bytecode verifiers', *Theoretical Computer Science* **298**(3), 583–626.

Kropf, T. [1997], *Formal Hardware Verification - Methods and Systems in Comparison*, Springer-Verlag, London, UK.

Lann, G. L. [1997], 'An analysis of the Ariane 5 flight 501 failure: a system engineering perspective', *IEEE International Conference on the Engineering of Computer-Based Systems* **0**, 339.

Leveson, N. G. and Turner, C. S. [1993], 'An investigation of the Therac-25 accidents', *Computer* **26**(7), 18–41.

MacKenzie, D. [2001], *Mechanizing proof: computing, risk, and trust*, MIT Press, Cambridge, MA, USA.

Melham, T. [1993], *Higher Order Logic and Hardware Verification*, number 31 *in* 'Cambridge Tracts in Theoretical Computer Science', Cambridge University Press.

Moore, J. S. [1989], 'A mechanically verified language implementation', *Journal of Automated Reasoning* **5**(4), 461–492.

Morgan, C. [1990], *Programming from specifications*, Prentice-Hall, Inc., NJ, USA.

Morris, F. [1973], Advice on structuring compilers and proving them correct, *in* 'Proceedings of the 1st annual ACM SIGACT-SIGPLAN symposium on Principles of programming languages', ACM Press, New York, NY, USA, pp. 144–152.

Mosses, P. D. [1996], Theory and practice of action semantics, *in* 'MFCS '96, Proceedings 21st International Symposium on Mathematical Foundations of Computer Science', Vol. 1113, Springer-Verlag, pp. 37–61.

Myers, G. J. and Sandler, C. [2004], *The Art of Software Testing*, John Wiley & Sons.

Necula, G. C. [1997], Proof-carrying code, *in* 'Proceedings of the 24th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Langauges', Paris, pp. 106–119.

Necula, G. C. and Lee, P. [1998], The design and implementation of a certifying compiler, *in* 'Proceedings of the 1998 ACM SIGPLAN Conference on Prgramming Language Design and Implementation', pp. 333–344.

Nelson, G. [1989], 'A generalization of dijkstra's calculus', *ACM Transactions on Programming Languages and Systems* **11**(4), 517–561.

Nelson, G. and Manasse, M. [1992], The proof of a second step of a factored compiler, *in* 'Lecture Notes for International Summer School on Programming and Mathematical Method', Vol. 88, Springer-Verlag, Secaucus, NJ, USA.

Newman, M. [2002], Software errors cost U.S. economy $59.5 billion annually, NIST assesses technical needs of industry to improve software-testing, Technical Report NIST 2002-10, National Institute of Standards and Technology.

Nicola, R. D. and Hennessy, M. [1984], 'Testing equivalences for processes.', *Theoretical Computer Science* **34**, 83–133.

Norrish, M. and Slind, K. [2007], *The HOL System: Description*.

Oliva, D. P. [1994], *Advice on Structuring Compiler Back Ends and Proving them Correct*, College of Computer Science, Northeastern University.

Oliveira, B., Cavalcanti, A. and Sampaio, A. [2002], Automation of a Normal Form Reduction Strategy for Object-oriented Programming, *in* 'Proceedings of the 5th Brazilian Workshop on Formal Methods'.
URL: *http://www.cs.kent.ac.uk/pubs/2002/1473*

Oliveira, M., Cavalcanti, A. and Woodcock, J. [2006], Unifying theories in proofpower-z, *in* S. Dunne and B. Stoddart, eds, 'UTP', Vol. 4010 of *Lecture Notes in Computer Science*, Springer, pp. 123–140.

Owre, S., Shankar, N., Rushby, J. M. and Stringer-Calvert, D. W. J. [2001], 'PVS system guide'. SRI International,November.

Page, I. and Luk, W. [1991], Compiling Occam into field-programmable gate arrays, *in* W. Moore and W. Luk, eds, 'FPGAs, Oxford Workshop on Field Programmable Logic and Applications', Abingdon EE&CS Books, Abingdon, UK, pp. 271–283.

Perna, J. I. and Woodcock, J. [2007], A denotational semantics for Handel-C hardware compilation, *in* M. Butler, M. G. Hinchey and M. M. Larrondo-Petrie, eds, 'International Conference on Formal and Engineering Methods 2007', Vol. 4789 of *Lecture Notes in Computer Science*, Springer, pp. 266–285.

Perna, J. I. and Woodcock, J. [2008], Wire-Wise Correctness for Handel-C Synthesis in HOL, *in* 'Seventh International Workshop on Designing Correct Circuits (DCC)', pp. 86–100.

Perry, D. and Foster, H. [2005], *Applied Formal Verification: For Digital Circuit Design*, McGraw-Hill Professional Publishing, New York, NY, USA.

Pfleeger, S. L. and Hatton, L. [1997], 'Investigating the influence of formal methods', *Computer* **30**(2), 33–43.

Polak, W. [1981], *Compiler Specification and Verification*, Springer-Verlag New York, Inc., Secaucus, NJ, USA.

Pu, G., Qiu, Z. and He, J. [2005], 'Integrating time and resource into *circus*', *Electronic Notes Theoretical Computer Science* **130**, 401–418.

Qin, S., He, J., Qiu, Z. and Zhang, N. [2002], Hardware/software partitioning in Verilog, *in* 'International Conference on Formal and Engineering Methods 2002', Springer-Verlag, London, UK, pp. 168–179.

Raje, S. [2004], 'Catching the FPGA productivity wave', *Electronic Design Journal* .

Roscoe, A. W. [1985], Denotational semantics for Occam, *in* G. Winskel, ed., 'Seminar on Concurrency, Carnegie-Mellon University', Springer-Verlag, pp. 306–329.

Roscoe, A. W. [1997], *The Theory and Practice of Concurrency*, Prentice Hall PTR, NJ, USA.

Roscoe, A. W., Goldsmith, M. and Scott, B. [1993], Denotational semantics for Occam II, Technical Report PRG-108, Oxford University Computing Laboratory.
URL: *http://web.comlab.ox.ac.uk/oucl/work/bill.roscoe/publications/47.pdf*

Roscoe, A. W. and Hoare, C. A. R. [1988], 'The laws of Occam programming', *Theoretical Computer Science* **60**(2), 177–229.

Sampaio, A. [1993], An Algebraic Approach to Compiler Design, PhD thesis, Programming Research Group, Oxford University.

Sampaio, A. [1997], *An algebraic Approach to Compiler Design*, World Scientific Publishing Company.

Silva, L., Sampaio, A. and Barros, E. [1997*a*], A normal form reduction strategy for hardware/software partitioning, *in* 'Formal Methods Europe 1997', Springer-Verlag, London, UK, pp. 624–643.

Silva, L., Sampaio, A. and Barros, E. [1997*b*], A normal form reduction strategy for hardware/software partitioning using Occam, Technical report, Federal University of Pernambuco.

Silva, L., Sampaio, A. and Barros, E. [2004], 'A constructive approach to hardware/software partitioning', *Formal Methods in System Design* **24**(1), 45–90.

Silva, L., Sampaio, A., Barros, E. and Iyoda, J. [1999], An algebraic approach to combining processes in a hardware/software partitioning environment, *in* 'Algebraic Methodology and Software Technology 1998', Springer-Verlag, London, UK, pp. 308–324.

Smith, M. A. and Gibbons, J. [2007], Unifying theories of objects, *in* J. Davies and J. Gibbons, eds, 'Integrated Formal Methods 2007', Vol. 4591 of *Lecture Notes in Computer Science*, Springer-Verlag, pp. 599–618.
URL: *http://www.comlab.ox.ac.uk/jeremy.gibbons/publications/uto.pdf*

Smith, M. A. and Gibbons, J. [2008], Unifying theories of locations, *in* A. Butterfield, ed., 'Unifying Theories of Programming', Dublin.
URL: *http://www.comlab.ox.ac.uk/jeremy.gibbons/publications/utl.pdf*

Srivas, M., Rueß, H. and Cyrluk, D. [1997], Hardware verification using PVS, *in* T. Kropf, ed., 'Formal Hardware Verification: Methods and Systems in Comparison', Vol. 1287 of *Lecture Notes in Computer Science*, Springer-Verlag, pp. 156–205.

Stark, R. F., Borger, E. and Schmid, J. [2001], *Java and the Java Virtual Machine: Definition, Verification, Validation with Cdrom*, Springer-Verlag New York, Inc., Secaucus, NJ, USA.

Stepney, S. [1993], *High Integrity Compilation : a case study*, Prentice-Hall.

Stepney, S. [1998], Incremental development of a high integrity compiler: experience from an industrial development, *in* 'Third IEEE High-Assurance Systems Engineering Symposium'.

Stepney, S., Whitely, D., Cooper, D. and Grant, C. [1991], 'A demonstrably correct compiler', *Formal Aspects of Computing* **3**(1), 58–101.

Susanto, K. W. and Melham, T. [2001], 'Formally analyzed dynamic synthesis of hardware', *Journal of Supercomputing* **19**(1), 7–22.

Thatcher, J. W., Wagner, E. G. and Wright, J. B. [1979], More on advice on structuring compilers and proving them correct, *in* 'Proceedings of the 6th Colloquium, on Automata, Languages and Programming', Springer-Verlag, London, UK, pp. 596–615.

Thomas, D. E. and Moorby, P. R. [1998], *The Verilog hardware description language (4th ed.)*, Kluwer Academic Publishers, Norwell, MA, USA.

Tremblay, J.-P. and Sorenson, P. G. [1985], *Theory and Practice of Compiler Writing*, McGraw-Hill, Inc., New York, NY, USA.

Woodcock, J. C. P. and Cavalcanti, A. L. C. [2004], A Tutorial Introduction to Designs in Unifying Theories of Programming, *in* E. A. Boiten, J. Derrick and G. Smith, eds, 'Integrated Formal Methods 2004', Vol. 2999 of *Lecture Notes in Computer Science*, Springer-Verlag, pp. 40 – 66. Invited tutorial.

Woodcock, J. and Cavalcanti, A. [2001], A concurrent language for refinement, *in* A. Butterfield, G. Strong and C. Pahl, eds, 'IWFM', Workshops in Computing, BCS.

Woodcock, J. and Cavalcanti, A. [2002], The semantics of *Circus*, *in* 'International Conference of B and Z Users on Formal Specification and Development in Z and B 2002', Springer-Verlag, London, UK, pp. 184–203.

Woodcock, J. and Davies, J. [1996], *Using Z: specification, refinement, and proof*, Prentice-Hall, Inc., Upper Saddle River, NJ, USA.

Wright, C. and Arens, M. [2005], 'FPGA-based System-on-Module Approach Cuts Time to Market, Avoids Obsolescence', *FPGA and Programmable Logic Journal* **VI**(6).

Young, W. D. [1988], A verified code generator for a subset of Gypsy, PhD thesis. Supervisor-Boyer, Robert S. and Supervisor-Moore, J. Strother.

Young, W. D. [1989], 'A mechanically verified code generator', *Journal of Automated Reasoning* **5**(4), 493–518.

Zeyda, F. and Cavalcanti, A. [2009], 'Mechanical reasoning about families of UTP theories', *Electronic Notes in Theoretical Computer Science* **240**, 239–257.

Author Index

He [2002], 20, 121, 189
Hierons et al. [2009], 13
Hoare and He [1998], 17, 23, 31, 32, 37, 38, 41, 44, 50, 52, 55, 58, 59, 68, 69, 72, 179, 180, 182, 193, 205, 247, 263, 302
Hoare et al. [1993], 123
Hoare et al. [2000], 19
Hoare [1983], 22, 23, 26, 37, 69, 99, 188
Hoare [1985], 14
Hoare [1990], 17
Hoffmann [1998], 15
Hudak et al. [1992], 37

I

IEEE [1993], 22, 30
ISO [1999], 22
Iyoda and He [2001*a*], 20, 145, 189
Iyoda and He [2001*b*], 20, 121, 145, 189
Iyoda [2007], 31, 144

J

Joost and Salomon [2005], 14

K

Kamat et al. [2009], 22
Kernighan and Ritchie [1988], 22, 98
Klein and Nipkow [2003], 15, 31
Kropf [1997], 21

L

Lann [1997], 13
Leveson and Turner [1993], 13

M

MacKenzie [2001], 13
Melham [1993], 31
Moore [1989], 15
Morgan [1990], 19, 105
Morris [1973], 17
Mosses [1996], 14
Myers and Sandler [2004], 13

N

Necula and Lee [1998], 16
Necula [1997], 16
Nelson and Manasse [1992], 116
Nelson [1989], 91
Newman [2002], 13
Nicola and Hennessy [1984], 30
Norrish and Slind [2007], 31

O

Oliva [1994], 15
Oliveira et al. [2002], 19
Oliveira et al. [2006], 181
Owre et al. [2001], 15

P

Page and Luk [1991], 23
Perna and Woodcock [2007], 30
Perna and Woodcock [2008], 31
Perry and Foster [2005], 21
Pfleeger and Hatton [1997], 13
Polak [1981], 15
Pu et al. [2005], 180

Q

Qin et al. [2002], 20, 189

R

Raje [2004], 14
Roscoe and Hoare [1988], 19, 30, 188
Roscoe et al. [1993], 30
Roscoe [1985], 30
Roscoe [1997], 30

S

Sampaio [1993], 18, 19, 116
Sampaio [1997], 18, 21, 121, 180
Silva et al. [1997*a*], 19, 190
Silva et al. [1997*b*], 19, 116
Silva et al. [1999], 19
Silva et al. [2004], 19, 190
Smith and Gibbons [2007], 181

Index