Temporal Reasoning About Robotics Applications: Refinement and Temporal Logic

Abdulrazaq Abba

Doctor of Philosophy

The University of York
Computer Science

October, 2020

Abstract

The challenges of verifying the behaviour of robotics systems has motivated the development of various techniques and tools for supporting the advancement and verification of robotics systems. This is due to the complex nature of verifying robotics systems as part of the category of hybrid dynamical systems that combine discrete and continuous parts. In contrast to the commonly-known computer systems, robotic systems operate in a physical, real-world environment that may include humans, which raises a reasonable question of concern about the safety of the systems. Currently, one of the promising solutions is effective, rigorous verification techniques and tools that verify and guarantee the safe operation of robotics systems [1] .

Along this line, formal methods provide mathematical models that support the development of rigorous verification techniques and tools. In this work, we use formal methods for the verification of temporal specifications of robotics systems. The process algebra *tock-CSP* provides textual notations for modelling discrete-time behaviours, with the support of various tools for verification. Also, *tock-CSP* has been used to give semantics to a domain-specific language for robotics, RoboChart. Similarly, automatic verification of Timed Automata (TA) is supported by the real-time verification toolbox UPPAAL that facilitates verification of temporal specifications using Time Computation Tree Logic (TCTL). Timed Automata and *tock-CSP* differ in both modelling and verification approaches. For instance, liveness requirements are difficult to specify with the constructs of *tock-CSP*, but they are easy to verify in UPPAAL.

In this work, we add a step forward in translating *tock-CSP* into TA to take advantage of UPPAAL. We have developed a translation technique and tool; our work uses rules for translating *tock-CSP* into a network of small TAs, which address the complexity of capturing the compositionality of *tock-CSP*. For the validation of our proposed contributions, we use an experimental approach based on finite approximations to trace sets. We consider trace semantics for validating the translation technique. Thus, we develop a technique for generating and comparing traces of *tock-CSP* and TA. In order to evaluate the translation technique and its corresponding tool, we use two forms of test cases: a large collection of small processes and case studies from the literature. We illustrate a plan for using mathematical proof to establish the correctness of the rules that will cover an infinite set of traces.

[1]In this work, we consider the definition of robotics systems in a broader sense as described in the IEEE standard 1875-2015, `https://ieeexplore.ieee.org/document/7084073`

Contents

List of Tables

List of Figures

Acknowledgement

Praise be to almighty God, Lord of the worlds, the compassionate, the merciful; I remain grateful for enriching me with health and courage to complete this challenging and interesting task.

I wish I have enough word to express my appreciation to my parents, my father Alh. Hassan Abba, my mother Izzatu Lawan; the people that hear the echo of silence thousands of miles away, and also the entire family for your prayers, encouragement and support, all I can say, may almighty reward you with the best of his reward.

I would like to thank my supervisors Ana Cavalcanti and Jeremy Jacob for their kind support, guidance, dedication and insightful discussions that shape the direction of this work. The completion of this work has been aided considerably by their dedication to help me complete this study.

I wish to express my gratitude to internal examiner Radu Calinescu, who remains in my Thesis Advisory Panel (TAP) since the beginning of this journey up to the ending. Your comments, discussions and corrections have helped a lot in improving this work. I would also like to thank my external examiner for his thoughtful questions, pleasant discussions and useful comments. I am grateful for all your supports.

Special thanks go to all my teachers, from pre-school to university, both formal and informal schools. It is your hard-work that raises me to this level. I consider myself very fortunate enough in getting to this ambitious terminal degree. Also, my special thanks to the Petroleum Technology Development Fund (PTDF) for financing this study. I am thankful for this opportunity.

I remain grateful to my grandfather, who initiates the vision of educating his heirs because of his belief in the power of education. His decision is one of the initial seeds of this PhD study, may his soul rest in perfect peace. I wish he is alive to witness this moment as part of his legacy. I inherit this belief in the power of education, and I will extend this legacy to the best of my ability. Education is the key to a good life, a solution to many problems and a source of hope to humanity,

I wish to express my gratitude to family, friends, office-mates, colleagues and many people that help me in various ways: advice, finance, hospitality, friendship and so on. I have been fortunate enough to be in a community of good people in the department, university and the city of York.

Finally, I would like to thank you, a brave reader, for your interest in this thesis and for taking the time to read through it, including this acknowledgement. I hope you will benefit from this work to expand your knowledge.

.

Declaration

I hereby declare that the contents of this thesis are the result of my own original contribution, except where otherwise stated. I am the sole author of this work and this work has not previously been presented for an award at this, or any other University. All relevant sources are acknowledged as references.

Chapter One

1. Introduction

One of the significant achievements of technology is the development of robotics systems; a category of systems that interprets the environment, takes action, moves (mobile robots) and performs a series of operations while avoiding harmful effects without human intervention. This includes the kind of systems that both interact with and learn from the environment as well as adjust their behaviour according to their current knowledge and interpretation of the environment.

The capabilities of these systems offer great promise in the development of technology, economy and improving life in general. Since robotics systems operate in the physical world and share an environment with human and other living creatures, one of the key issues of these systems is their ability to avoid harmful behaviour. In circumstances where safety is critical, it is therefore imperative that these systems are developed to minimise the possibility of undesirable behaviour. Currently, ensuring the behaviours of these systems are both correct and safe remains an open question of concern [3, 4].

In the recent past, various techniques and tools have been developed to support the verification of robotics systems. In this regard, one of the promising approaches is applying rigorous validation and verification techniques to the control software. However, the effective methods, techniques and tools for checking the behaviour of robotics systems is one of the conundrums of software engineering [5]. The motivation behind this work is to discover ways of improving the verification techniques that will support the development of better robotics systems, thus increase confidence in their trustworthiness.

In the aim of addressing this problem, we take a step forward for improving the techniques and tools used for checking the temporal specification of the control software for robotics systems. In this work, we consider temporal reasoning, and we describe a formal translation of tock-Communicating-Sequential-Processes (*tock-CSP*) [6] into Timed Automata (TA) [7] that facilitates using the Timed Computational Tree Logic (TCTL) in the verification of temporal specifications, with the automatic support of UPPAAL [7].

This chapter introduces the research conducted in this work. It begins with discussing the main motivation of the work, including the four directions that motivate the work: Software Engineering for Robotics in Section 1.2, Domain Specific Modelling Languages (DSML) in Section 1.3, formal methods in Section 1.3.1 and then Temporal Reasoning in Section 1.3.2; followed by Section 1.4 that describes the contributions of this work. Lastly, the chapter concludes with an outline structure of the rest of the document in Section 1.5.

1.1. Motivation

The current technology is highly dependent on software systems. Nowadays, software systems play a vital role in the operation and control of systems; including robotics. Software systems open a door for considerable achievement in the area of robotics. In recent years, robotics systems have been recognised as one of the most important technologies, as nowadays they are becoming acceptable in almost every sector: industry, school and home [8, 9]. In industry, for example, robotics systems have successfully reduced human labour in various sectors, such as automotive and electronics. In the same vein, it has been reported that robotics systems are significantly increasing productivity, and there are more predictions for this technology's continuous rise in future years [10, 11].

The capabilities and flexibility of robotic and autonomous systems (RAS) make them suitable for handling both difficult and harmful tasks [12]. They are also suitable for use in hazardous environments such as in rescue missions, mining, chemical and explosives detection [13, 14]. Additionally, they also support life and improve welfare, for instance, in assisting aged and disabled people to live independently [15].

This research is important because in this current age of technology robotics systems are moving from a controlled industrial environment to the world for universal commercial use. Thus, the need for safety becomes a crucial point of concern because these systems will be operating and interacting with the physical environment that may involve sharing an environment with humans. Importantly, the environment may include critical equipment, or the behaviour of the robotics system may involve operating critical equipment. Ensuring the safety and correctness of their operation is essential for the continuation of their success.

This can be achieved with rigorous verification and validation (V&V) techniques that check whether a system satisfies its specification correctly, both logically and temporally, before launching it into the full operation for universal purpose. For years, these V&V techniques have been used as an essential part of the Software Engineering (SE) tool kit [16].

The main objective of the research is extending and complementing the existence research for addressing the problem of verifying temporal specifications of robotics applications. Thus, we study the available resources for verifying temporal specifications of robotics applications, where we select *tock-CSP* and UPPAAL. This is because *tock-CSP* captures the generated semantics of the RoboChart models, which will remain consistent despite the evolution of the structure of the RoboChart and its tool RoboTool. This choice will have an additional advantage of enhancing the application of this research to enable other related researches based on *tock-CSP* to use our work.

1.2. Software Engineering for Robotics

In this technological era, advancement in hardware development increases the demand for software that operates the hardware correctly. Likewise, robotics increases the demand for support from SE for developing quality applications for robotics; for complementing the robotics hardware with efficient software, which operates and controls an autonomous system safely and correctly [17,18].

The development of a software system for operating and managing robotics systems mostly involves various areas of expertise. Although the expertise involved depends on the type of system and the targeted operating environment, typically this includes SE, Artificial Intelligence (AI), Electronics, Mechanics, and Human-Computer Interaction (HCI). Combining these kinds of expertise to develop a sound software system remains a challenging task.

One of the target goals of SE is providing an abstraction that supports smooth collaboration among the various experts involved in the development of robotic software systems. This abstraction provides an effective separation of concern that decomposes a system into modules of various concerns each for a specific expertise [17]. SE techniques, such as Component-Based Software Development (CBSD), provide useful means for breaking down a complex robotics system into smaller units that are developed and managed independently [19,20]. The abstract descriptions of each unit are used to describe and design a complete system.

Also, in some cases, software systems for RAS are developed with partial information of both the system behaviour and its operating environment because robotics systems perform a series of continuous actions with partial feedback and uncertainty about the effect of each action. Additionally, in some cases, these systems are expected to operate in a dynamic environment that evolves and changes at any given time during operation. Importantly, this poses the need for exercising extreme caution when putting the system into operation.

Further, the design and development of robotics systems involve handling a variety of unrestricted requirements due to the nature of their operating environment, which comprises mostly open-ended unrestricted environments. More often, these requirements are subject to change during the system operation. These concerns create many complexities in the operation of the systems. However, these complexities can be handled and managed effectively using well-developed systematic techniques for rigorous verification.

The field of SE has a collection of well-established systematic concepts and approaches for incorporating various software aspects properly at each phase of software development. These concepts have been used successfully to develop software systems for multiple fields, such as nuclear factory, medicine, archaeology, similar to the expectation that robotics are expected to be used in multiple fields. These are also applicable to robotics, specifically in improving the quality of the robotics systems by providing better techniques for developing software systems that improve safety of robotics, while still improving both the performance and the complexity of verifying robotics applications.

In most cases, a robotics system consists of components that operate concurrently while processing a large amount of accumulated data in real-time and reacting to the operating environment within the budgeted (scheduled) time due to real-time constraints. As a result of this, most of the research focuses on improving performance and addressing complexity. Thus, less attention is paid to other important quality issues such as verification, safety, reliability and maintenance [21].

In this case, suitable systematic techniques of SE for combining each aspect of the software properly at each developmental phase will play a vital role in balancing each aspect that will enable the production of quality software; the kind of software that is safe for use in operating robotics systems. This will improve the quality of robotics systems considerably by making them trustworthy systems. Additionally, DSML will improve further by narrowing down the scope of robotics systems and reducing the complexity of both handling the requirements and development of the systems.

In this work, we consider the definition of robotic in broader sense as described in IEEE Standard 1872-2015 [2]. As described below:

> An agentive device in a broad sense, purposed to act in the physical world in order to accomplish one or more tasks. In some cases, the actions of a robot might be subordinated to actions of other agents, such as software agents (bots) or humans. A robot is composed of suitable mechanical and electronic parts. Robots might form social groups, where they interact to achieve a common goal. A robot (or a group of robots) can form robotic systems together with special environments geared to facilitate their work.

1.3. Domain-Specific Modelling Languages (DSML)

Considering the achievement of software abstraction and Model-Driven Engineering (MDE) in SE, there are several proposals for developing an effective DSML for supporting Robotic Software System Development (RSSD) [22–24]. A DSML that is well equipped with desirable features for developing reliable robotics systems; with comprehensive formal techniques and tools for exhaustive verification and validation techniques [1].

The concept of modelling systems before their implementation remains an interesting concept that is pushing SE forward. DSML adds a step forward in providing effective facilities for defining an abstract system that hides unnecessary details. This concept of DSML is visible in three dimensions: effective separation of concern that reduces complexity, eases prototyping and overall speeds up the development process [24,25].

Also, the concept of using DSML has been successfully tested for use in the development of software systems in various fields that produced a good result in reducing domain complexities for an appropriate suitable solution to a specific problem. Additionally, the combination of the various areas of expertise involved in robotics makes SE and specifically DSML suitable candidates for use in the development of robotic software systems [24].

In the available literature, many DSMLs have been developed and used in the development of robotics systems. Most of these languages consider providing facilities for improving capabilities of robotics systems, such as motion, recognition and planning. Among the available DSMLs in the literature, only three DSMLs considered

[2]`https://ieeexplore.ieee.org/document/7084073`

using formal techniques for verification. These are GenoM [26], DSML for Adaptive Systems [27], and RoboChart [13]. Additional details are provided in Chapter 2.

In the case of GenoM, it was initially developed as a tool for generating models from a combination of models and code, with no basis for using formal techniques. Use of formal techniques was added as an extension for improving its strength in verification. In comparison, DSML for Adaptive Systems was developed as a textual language with a very narrow scope for modelling the adaptation logic only. RoboChart was developed as a graphical language with a good foundation for using formal techniques, since its inception. Thus, RoboChart provides formal notations that are suitable for developing formal models.

In short, complementing a robotics DSML with formal techniques has provided good additional facilities for improving the quality and safety of robotics systems; more specifically, in improving the verification aspect that will reduce considerable defects.

1.3.1. Formal Methods

The use of mathematics for analysis and verification is the basis of formal method (FM), which enables using well-established theorem proving techniques and model checking for formal verification. Using FM has motivated the production of a broad spectrum of mathematical techniques for constructing and analysing models of computer systems with logical reasoning. This leads to a notable contribution to quality improvement, which has been reported as one of the main strengths and contributions of the FM in software development. Significantly, this concept improves quality and productivity, as it is widely recommended that formalisation is used for designing trustworthy systems [28–31].

In designing systems (such as robotics systems), FM emphasises the concept of creating abstract models to specify the required characteristics of systems. The abstract models are subjected to rigorous consistency checks to determine whether they meet the intended requirements. As intended, formal techniques have been successfully tested for Verification and Validation (V&V) in various industrial sectors, such as transport, business and critical systems. They have been used in various types of projects for both hardware and software. For example, FM has been used successfully in designing and verifying microprocessors, internet protocols and scheduling [21, 29, 32]. This success tallies with the expectation that robotics systems are use for various services, ranging from toys for fun to large critical industries like nuclear systems.

Using formal verification serves as a better alternative to testing in ensuring the correctness of a system. This is due to the advantage of formal validation in detecting bugs at the early development phase, at the design level, rather than at the implementation level, which is more costly [21]. This advantage plays a significant role in reducing overall project cost and time. This contribution has been achieved with the use of formal notations such as CSP [33], Z [34], and Alloy [35] that have been used for system modelling and verification using formal techniques.

In addition, the approach of using a formal method is suitable for complementing and improving the currently used popular approaches for testing and simulation. For a long time, simulation and testing have been used in software development in various fields with a satisfactory result. However, both of these approaches are incomplete because they only check a specifically formulated scenario. It is clear that a collection of formulated scenarios or test cases only count for a fraction of the targeted operating environment [21]. This can be improved significantly by using formal techniques for checking all possible behaviour of the system to establish whether a system satisfies its specifications. Usually, this is achieved using well-established theorem-proving techniques and model checking for formal verification.

1.3.2. Temporal Reasoning

In contrast to the previous sections that discussed techniques and tools, this section focuses on the role of time in organising the flow of activities to define the behaviour of a system. Timing is an essential component of robotics systems; it is used to assemble and schedule a series of activities that accomplish a mission correctly. Therefore, investigating the correctness of system behaviour with respect to the flow of time is essential in the overall system behaviour, especially the robotics system that operates in real-time.

Depending on the nature of a system, analysing system behaviour in terms of a sequence of events could be appropriate for reasoning about untimed safety. However, in the case of timed sensitive systems, especially, the kind of systems that operate in real-time, reasoning about time is essential for its safety and correct operation. The behaviour of timed sensitive systems needs to be specified at the level of timed detail, which takes into account timing, scheduling, delay and deadline [36].

There are various approaches to modelling temporal specification. For instance, in refinement modelling approach such as in CSP notations, the facilities for addressing the timing specification was provided by extending the existing untimed notations. This is achieved by providing additional constructs for capturing time specification, such as *tock-CSP* [6] and Timed CSP [37]. In the same manner, Circus Time [38] is an extension of Circus with the notion of time. Both of these notations use the discrete approximation of time, which is appropriate at the level of computer application. However, there are specifications that can not be specified and verified in the discrete-time model.

Another popular approach of modelling temporal specification is using temporal logics that provide different constructs for verifying temporal specifications. Both modelling approaches of refinement and temporal logic are powerful approaches for model checking systems [39]. The refinement approach models both the system and its specifications with the same notation [6,37]. Temporal logic enables asking whether a system is a model for a logical formula of the specification ($system \models formula$) [40].

In the literature, Lowe has investigated the relationship between the refinement approach (in CSP) and the temporal logic approach [39]. The result of Lowe's work shows that, in expressing temporal logic checks using refinement, it is necessary to use the infinite refusal testing model of CSP. The work highlights that capturing the expressive power of temporal logic in specifying the availability of an event (liveness specification) is not possible. Also, due to the difficulty of capturing refusal testing, automatic support becomes problematic with the previous version of the CSP supporting tool Failures-Divergence Refinement (FDR) supports refusal testing, but not its recent efficient version [41].

There are three classes of specifications that can not be express with simple refinement check. Lowe's [39] proves that simple refinement checks cannot cope with these three operators: *eventually* ($\diamond p$: p will hold in some subsequent state), *until* ($p\mathcal{U}q$: p holds in every state until q holds) and *negation* ($\neg(\diamond p)$: p will never hold in the subsequent states). All these three operators express behaviour that is captured by infinite traces. In this work, we introduce a translation tool, presented here, which will facilitate using the resources of temporal logic (and Uppaal for automatic support) in checking *tock-CSP* models, particularly specification that are difficult to specify with refinement model.

In plain English, example of these specifications are:

1. The robot will eventually reach the target goal.
2. The robot remains in a safe state until the hazard disappear.
3. The robot will not cross the barrier until the barrier open.

Time is in continuous form, but the basis of computing happens to be in digital form, which is in discrete form. Real life is a continuous flow of events in continuous form, and correspondingly a real-environment evolves continuously with real-time. Therefore, a specification of a system that behaves and interacts with a real environment needs to be more appropriate in the hybrid form; a combination of continuous and discrete form. Precisely, in the case of reasoning, it has been reported that, the number of reachable states in modelling a system using a continuous-time model is always greater than or equal to the number of reachable states in modelling the same system using a discrete-time model [36,42,43]. This shows that in using discrete-time models alone, it is impossible to reason about the unreachable states. A discrete model of time has a fixed time interval (time unit). If the interval is too coarse, some reachable states will be missed. Thus, it is not possible to draw a complete conclusion about the behaviour of a system without considering the unreachable states.

On the one hand, it can be argued that finding a good enough granularity can provide an equivalent correct result, which is similar to modelling the system with a continuous-time model. On the other hand, there is no known simple way of finding enough granularity that is good enough to provide a similar correct result that is equivalent to the continuous-time model. Within the studied literature, there is no well-defined process of finding an optimal granularity that is comparable to a continuous-time model. It has been mentioned that it is likely that the process of finding the optimal granularity is as complex as solving the whole problem using the continuous-time model. In some cases, the procedure could be worse than using the continuous-time model. This is because the finer the granularity, the more it reduces the efficiency of the verification process. In contrast, it is more likely to increase the complexity of the verification techniques [5,36].

In the literature, there is an interesting concern about handling temporal specifications. For instance, all the timed models of the relevant DSMLs found in the literature have considered using discrete-time approximation for modelling and verification of temporal specification. In conclusion, most of the modelling notations and constructs used a form of discrete time in verifying temporal specifications. Considering the safety concern for real-time systems and the uncertainties associated with discrete approximation, as highlighted above, there is a need for a better alternative approach for improvement.

1.4. Contributions

The goal of the proposed work is to facilitate verification of temporal specifications of robotics systems using Uppaal, which provides facilities for supporting reasoning with temporal logic, especially for the verification of safety properties. These provided facilities can complement the limitations of refinement approach in expressing temporal specifications. Previous research [39] has established that reasoning with simple refinement checks has a limited scope in expressing temporal specifications; for instance, expressing liveness specifications is beyond the scope of refinement checks [39].

In the literature, many techniques and tools have been developed for improving robotics applications. There is a clear indication of less concern regarding the verification of temporal specifications. We discuss this in more detail in the literature review (Chapter 2). This work focuses on improving the verification tools by translating *tock-CSP* into TA that will facilitate reasoning with temporal logic in the verification of temporal specifications.

The concept of using automata for modelling system behaviour has been extended with the notion of a clock, which is capable of capturing time using an extension of automata called Timed-automata. System behaviour is modelled as a network of timed automata, which enables automatic verification of real-time systems with a well-known tool called Uppaal model checker [7]; an integrated tool for modelling and verification of real-time systems. The tool has been developed with a promising formal approach for verifying system properties, including temporal specifications. Techniques have been developed for translating RoboChart models into *tock-CSP* that facilitates verifying system specifications using a refinement approach with the automatic support of a verification tool called FDR [41]. FDR is an automatic verification tool for CSP that also supports *tock-CSP*.

These issues raise a fundamental question in using FM for verifying robotics applications. The most fundamental question that emerged from early exploratory work was how to improve formal techniques of verifying temporal specifications that are compatible with the existing resources of developing robotics, especially the ones that have a formal basis, such as RoboChart.

The methodology we consider for this work is model transformation. To answer the research question, we plan to improve the resources used for checking the temporal specification of the control software for robotics applications. The aim of this research work is to investigate suitable ways of using UPPAAL to verify *tock-CSP* specifications using the resources of temporal logic provided in UPPAAL.

To address the research question, we setup the following objectives:

1. The first goal is to study the available resources for verifying temporal specifications of robotics applications.

2. In the second goal, our work will add a step forward by developing techniques for automatically translating *tock-CSP* models into UPPAAL models; a kind of translation that maintains the existing semantics of *tock-CSP* in generating a correct and suitable TA model as input to the UPPAAL model checker that will facilitate verifying temporal specifications using TCTL. Therefore, existing works based on *tock-CSP*, for instance, RoboChart, will benefit from the facilities of two well-known verification tools FDR and UPPAAL. At the time of conducting this work, RoboChart is still evolving but the technique will be applicable to RoboChart because its semantics is in *tock-CSP*, and other related work around *tock-CSP*. Also, these techniques can serve as a link between the two popular model-checking tools FDR and UPPAAL, and a means of combining the two tools for system verification, with the best of the available resources, like improving performance in the verification stage.

3. The third goal is to automate the translation technique with a tool that will improve the usability of the work by making it easy to use. As a result, a significant step forward will be made in combining the facilities of both modelling approaches; as well as improving our understanding of the complex relationship between *tock-CSP* and TA, and also their modelling approaches for refinement and temporal logic, respectively.

4. The fourth goal is to sketch a mathematical proof that will provide additional justification for the correctness of the translation technique.

1.5. Structure of this Thesis

The rest of the document is structured as follows. Next, Chapter 2 describes the context of the work. We discuss the background study of the relevant DSML and tools for robotics applications that are available in the literature. The chapter begins with an overview of software development for robotics systems and discusses the role of DSML in developing software for robotics. The chapter proceeds with a discussion of interesting desirable features for a suitable DSML for robotics. Additionally, the chapter presents relevant DSMLs from the literature, we discuss and compare their available desirable features and also provide a brief account of using formal techniques in the verification aspect of the studied DSMLs. We consider RoboChart in this work; thus a separate discussion for RoboChart is added to provide a more detailed discussion of its constructs. Lastly, the chapter discusses relevant formal techniques and tools that are suitable for the verification of robotics systems; specifically CSP, its timed versions: *tock-CSP* and Timed CSP, its supporting tool FDR, then also Timed Automata (TA) and its supporting tool UPPAAL. In short, we describe how the literature sets a precedent and opens questions for the rest of the work.

In Chapter 3, we present the technique we have developed for translating *tock-CSP* models into suitable TA for UPPAAL model checker. We begin with presenting a BNF for the *tock-CSP* that describes the constructs of the *tock-CSP* we consider in this work. We describe a technique for translating the constructs of *tock-CSP* into TA. Examples are provided for more detailed illustrations both in Chapter 3 and Appendices. However, the main contribution of this chapter is providing a mechanism for translating *tock-CSP* into TA, whereby we can use temporal logic in the verification of the translated behaviour of *tock-CSP*.

In Chapter 4, we describe the approach we consider in evaluating the translation technique. We describe the automation of the translation technique, as well as another technique we developed for evaluating the translation technique. We use two categories of test cases: a large collection of formulated processes that capture interesting cases and a list of systems from the literature. We illustrate our plan of using mathematical proof to improve the justification of the translation technique.

Finally, in Chapter 5, we provide a general summary and conclusion of the work. The chapter begins by summarising the thesis, then highlights recommended directions of future work for expanding the research.

Chapter Two

2. Domain-Specific Modelling Languages and Tools for Robotics Applications

This chapter discusses relevant Domain-Specific Modelling Languages (DSMLs) and tools that are developed specifically for robotics systems that are available in the literature. The chapter is divided into nine sections. Section 2.1 traces back the concept of software development and then narrows down specifically to Robotics Software System Development (RSSD), which includes methods, techniques and approaches for RSSD. Also, we provide a brief account of both academic and industrial perspectives of RSSD at the end of this section. In Section 2.2, we describe a DSML and its desirable features. Section 2.3 presents the relevant DSMLs and compares their desirable features. In Section 2.4, we report on the formal verification and validation in DSML for robotics. Section 2.5 discusses RoboChart and its constructs in more details. In Section 2.6, we provide an account for automatic verification and validation in the studied DSMLs. Section 2.7 discusses *tock-CSP* and its supporting tool the FDR model checker. Section D discusses the UPPAAL model checker with its modelling language Timed Automata. Finally, in Section 2.9, we conclude the chapter.

2.1. Robotics Software System Development (RSSD)

In the early age of technology, manual [3] robotics systems were developed for industrial purposes. As the computing technology evolved, automatic robotics increasingly penetrated industrial environments. With the success of control software, industries have being automating their activities using robotics technology. This success is significantly enhancing industrial productivity because robotics systems facilitate continuous operation with fewer interventions [44].

However, an initial problem arises with the operating environment because most of the industries were not planned for automation, especially in the design aspect. Also, industries have inherent complexity related to their nature and settings, such as dynamic environment, real-time operation and associated uncertainty in their daily operations. In addressing this problem, the development of robotic systems was adjusted to match these industrial complexities. This brings about additional constraints in the development of robotic systems, which leads to various additional complexities, such as extra components: controllers, sensors and actuators. Usually, these components are arranged in a complex architecture that fits the purpose of the developed system according to the structure of the environment. Additionally, due to the complexity of designing good systems, the controls and communications are tightly coupled with the physical configurations of the system and the operating environment. Overall,

[3] Manual robot are operated directly by human, using joystick or other control devices for its operation with no stored software. For examples, exoskeleton robot, bulldozer and rock drilling.

this produces a complex system that can only be understood and programmed by experts [45].

The initial practices of programming robotics systems were manual processes. For instance, one of the commonly known approaches is Online Programming (OP) - programming a robotic system in the presence of its actual hardware component, which is accompanied with a programming device called a teach pendant. The teach pendant provides configuration information such as positioning and speed of the hardware that is used for programming the system. The configuration information is unique to each robot, not transferable to another robot or device. The approach of online programming facilitates robotics applications for a range of tasks, from simple to complex tasks; that completely depends on the skills of the operator, which limits its capability [45,46].

Advancement of microprocessors brings robot programming closer to computer programming. This achievement sets up the evolution of offline programming using software tools, which does not require the presence of physical hardware to program the robotics system. The robotics system is only needed for testing and evaluation. The implementation stage is supported by computer-aided design (CAD) tools for modelling both the robot and its operating environment, as well as its configuration. With the increasing success of offline programming, simulation becomes the most common base for testing the implementation [45,46].

Subsequent advancements in programming and supporting tools add a step forward in developing robotics application, which reduces the complexity, and makes programming easier. Also, the programming becomes closer to the artwork that is driven by creativity rather than skills because the majority of the hardware complexity becomes hidden from the programmers. However, another problem arises with the supporting tools having been created based on demand with neither concern for interoperability with other components, nor flexibility for reuse in some other related systems and platforms. This problem caused most of the robotics software projects to be developed as an independent project, that mostly begins entirely from scratch. In some cases, a large amount of time is spent on creating infrastructures of a project rather than the actual work of developing the system. Similarly, all the remaining stages, such as verification and validation (V&V) become an improvised task which also requires a large investment, as there is neither method nor tools to guide or support the V&V stages [46].

In the literature, there were calls for supporting the process of developing robotics systems with better development tools, which brings about the establishment of various projects such as BRICS [47], RoSta [48], OPRoS [49] and Orocos [50]. Best Practice in Robotics (BRICS) focused on structuring, standardising and formalising the development process of the robotics system. Robot Standards and Reference Architectures (RoSta) focused on providing standardisation for improving the concept of reusing robotics components and architectures. Open Platform for Robotic Service (OPRoS) focused on standardising CBSD for robotics systems. Open Robot Control Software (Orocos) focused on developing a standard framework and repository of robotics tools and components. These projects improved the RSSD, but other problems persist, and new ones emerge.

One of the problems that persist is the lack of a uniform programming platform. Unlike general computer programming where programming is independent of the hardware platform, the case of robotics systems is different because most of the robotic manufacturers developed their tools and programming languages for their own robotic systems [51]; for example, RAPID for ABB [52], KRL for Kuka [53] and Karel for Fanuc [54]. In some cases, manufacturers combine multiple general programming languages for programming their robotics system, which facilitates utilising the best-provided resources from the available programming languages that are suitable for programming robotics systems. This enables the best features of each language to be used for achieving different goals at various stages of the development; for example, Verilog or VHDL for Hardware, C/C++, Assembly language or Java for Firmware, Matlab, Octave or Simulink for controls and Python or C/C++ for cognition.

Consequently, initial attempts at addressing the uniformity of programming platforms lead to the development of general programming languages and tools for robotics systems; for example, ROBOTC [55], Visual Programming Language (VPL) and Urbiscript [56]. Additionally, tools were developed that targeted general platforms, such as Robot Operating System (ROS), Urbi and Open Robot Control Software (OROCOS). ROS [57] is a set of software libraries and tools for building robotic applications, ranging from simple drivers to a complete system. Urbi is an open-source framework for complex systems, mainly used to orchestrate components of a system [58]. OROCOS [50] is a component-based framework for creating robotics applications using modular configurable software components.

In a similar manner, some manufacturers take a different approach of providing configurations or software wrappers that enable using the general languages for programming their system. However, these wrappers only provide access to a partial functionality of the system with limited performance that limits the capability of programming the systems [59].

From the literature [60], we find that many concepts, techniques and frameworks have been developed for supporting specific aspects of robotics, which do not integrate with one another. This leads to another call for developing an integration umbrella for the large collection of tools developed for robotics systems. As a result of this, the Robotics Domain Task Force (Robotics DTF) [61] was established to support the integration of robotics components using the OMG standards. One of its notable contributions was developing the RTC specification [62]. RTC specification provides an integration point for independently developed components that are developed based on the provided standard specifications. Therefore, components integrate with one another openly if they are developed according to RTC specification.

To conclude this part, the above-listed works contribute to the advancement of RSSD. But some of the complexities persist, for instance using the traditional programming approaches, mainly coding at a low level, which happens to be unsuitable for effective analysis of large and complex programs like robotics applications. This is due to the lack of clear documentation, rigid design decisions that are difficult to maintain and lack of opportunity for reusing the components. These are among the significant issues for robotics and other large software systems. Furthermore, integrating code is another common problem, which is commonly achieved by reimplementing the glue logic code, based on the flexibility and capability of the available middleware, rather than developing a well-designed system with the suitable building blocks. This practice has a high tendency of wasting effort, resources and skills. Besides, the overall ad-hoc approach has no guarantee of producing correct and reliable systems [60].

Currently, there is increasing acceptance of robotics systems for domestics and other general uses, which raises the need for their safe operation. This is particularly very important for systems that operate in an unpredictable dynamic environment, including a critical environment. In this case, a provision of suitable techniques and tools for verifying the behaviour of RAS will be an invaluable support to the robotics community. This is especially useful if the techniques and tools are based on established SE principles, such as MDE, DSML and Formal Method that will make them compatible with the existing resources. In the next section, we are going to discuss desirable features for DSML that will support the development of robotic applications.

2.2. Features of DSMLs

This section discusses a selection of desirable features of DSMLs for developing robotics applications. The features are categorised into four sets, according to the aspect of software development to which they are relevant: modelling, reasoning, architecture and supporting tool. First, the modelling category consists of four items: time, probability, environment and API. Second, the reasoning category consists of three items: formal semantics, techniques for both validation and verification and facilities for simulation. Third, the system category consists of three items: platform independence, behavioural modelling and architectural style. Finally, the supporting tool category consists of IDE, framework, executable generation and accessibility.

2.2.1. Modelling

Time Here, we observe the existence of modelling constructs to support the definition of temporal specification in a system. Although it is possible to formalise a model of a system by abstracting the time aspects, for control systems in general, and robotic systems in particular, time is an important aspect. Most systems have a time cycle, time budgets and deadlines for operations as an important part of the system design. Explicit time modelling incorporates timing constraints that facilitate control, analysis and verification with regard to real-time behaviour and process scheduling [36].

Probability An existence of modelling constructs for describing system behaviour in handling uncertainties during its operation. In the case of uncertainty, handling decisions in an autonomous system is inevitable; therefore, modelling probability becomes an interesting concern in describing how a system handles uncertainties in making decisions. This provides a model of the randomness to which the system is exposed during its operations with respect to its operating environment [63]. Therefore, overall a DSML should provide facilities of modelling a system behaviour precisely according to the weight of various dynamics and events in the environment as well as other possible actions.

API Provision of facilities for defining, accessing and reusing generic operations that are common to robotics systems; for example, in the case of mobile systems move and stop are common behaviours, though it depends on the type of hardware platform. These operations are normally provided in accessible libraries that can be used and reused in developing robotics applications; for instance, RIPE [64], MRROC+ [65] and the Robotic Platform [66]. Also, flexible robotics API can be extended to create a custom function in an application. An additional advantage of using API is relieving software developers from understanding the low-level control of robotic hardware, which allows programmers to focus on application development. Facilities for API increases support for Component-Based Software Development (CBSD) using functional components with well-defined interfaces, which significantly increases both quality and productivity in using the DSML [67].

Environment A construct for modelling an operating environment that provides an apparent view of the system behaviour in relation to its environment. This is very important because robotic systems interact with their environment, taking input and reacting to an event using suitable action or set of actions. This plays an essential role in verifying system behaviour in various states of the operating environment [24].

2.2.2. Reasoning

Formal Semantics Provision of formalisation techniques that provide a precise mathematical specification of a system in an automatable way. This facilitates the use of formal verification techniques and tools for both verification and analysis. The advantages of these facilities includes eliminating ambiguity that is associated with both the natural language and graphical language, in addition to facilitating precise integration between various systems and components that have formal specifications [37].

Verification and Validation Techniques Provision of systematic V&V techniques, more specifically techniques that enable automated formal V&V. Provision of automatic formal V&V provides an easier way of establishing the existence or absence of specific behaviour, both good behaviour and harmful one. This is a valuable support for making the right decision about the correctness of a system [68].

Simulation Provision of facilities for testing the behaviour of an implementation of a system in a model of real-world settings. This feature has at least three advantages. First, simulation provides an easy way of exploring and experimenting with the implementation in control settings. Second, it enables both testing and verification of algorithms independent of a hardware platform. Lastly, this feature reduces deployment complexity and overall increases productivity and accuracy [69].

2.2.3. System Architecture

Platform Independence Eliminating tight coupling between the developed model and other generated artefacts of the targeted execution context. This makes it flexible for reusing the generated artefacts in multiple contexts. Also, this facility has an additional advantage of making it easier to use existing tools and resources when using the DSML; for example, libraries, runtime environment, platform and frameworks [24].

Behavioural Modelling The capability of a DSML in expressing low-level detail behaviour of a system. This includes the available control and flexibility that a developer has on describing system behaviours and processes [70].

Architectural Style Constructs or structures that the language uses to organise the system behaviour and its components; for example, the commonly known modular design that uses modules to organise related operations, data and events [71].

2.2.4. Tool Support

IDE An available tool either textual or graphical that provides a suitable environment for modelling and development. A good DSML should have a tool to support developers in the correct usage of both abstract and concrete syntax of the language [47].

Framework This is used to develop the supporting tool and the language itself. For example, Eclipse Modelling Framework (EMF) is commonly used to support DSML. EMF is one of the most popular frameworks that has been used to develop many tools for supporting DSMLs [72].

Executable Generation Part of our interest is to consider DSML that has artefacts in the form of executable logic, code or models. This can be an executable logic for direct use either in the hardware or simulation. A code can be in another language that can be compiled to generate the executable logic. Alternatively, the artefacts can be a model, in which the code development should be carried out separately [73].

Accessibility A description for accessing the tool and the language itself. Some DSMLs are provided with completely free access, and some are provided for a fee. For example, simulation facilities in MATLAB and Octave are examples of paid and free facilities, respectively. Some DSMLs combine both types of access depending on the purpose of research, study, testing, personal or commercial, in addition to updated guides such as documentation, tutorials, manuals and examples [24].

The features mentioned above are the desirable features of DSML for robotics. The next section discusses sample DSMLs with respect to these features. We consider each of the above desirable features in selecting and presenting the studied DSMLs for robotics found in the literature.

2.3. Sample DSMLs

In this section, nine DSMLs for robotics are presented. In selecting the languages to be presented here, one of the considerations is the availability of recent works that indicates active support and continuous development. Secondly, the list is intended to cover various approaches to system modelling.

There are other interesting languages that were not included in this list for various reasons because many DSMLs have been developed for robotics, as listed in the literature [22, 24]. Examples of other interesting DSMLs include ALFA [74], AMARSi DSL [75], rFSM [76], GSRAPID [3] and GRL [77].

ALFA DSML was developed with reactive control mechanisms for the development of autonomous mobile robotic systems. ALFA was not developed as a graphical modelling language. The language is old with no active support because the last published information dated back to 1991 [74]. AMARSi DSL was developed with a narrow scope and restricted application on modelling complex humanoid movements. MAESTRO [78], GSRAPID [79] and GRL [80] are examples of DSMLs that can continue enriching the list, but the list has to stop at a certain point.

This section is divided into two parts. The first part presents a comparison of the selected DSMLs based on the features highlighted in the previous section. A summary of the comparison is provided in Tables 1-4. The second part presents a brief description of each of the studied DSMLs. The description begins with a brief introduction of RoboChart here and more detail in Section 2.5 because RoboChart is the selected language for this work.

Tables 1 - 4 present a general comparison of the nine DSMLs listed in this study.

2.3.1. RoboChart

RoboChart is an evolving DSML developed with the aim of improving the safety and quality of robotic software systems, as well as reducing the complexity of verifying the behaviour of robotics systems. RoboChart is currently being developed as a formal graphical modelling language generating a formal specification of graphical models. This provides a good facility for using theorem-proving tools for automatic verification as well as simulations [81].

Modelling RoboChart provides notations for modelling system behaviour with explicit constructs for modelling time, such as specifying time budget and deadlines. For example, the primitive $wait(d)$ specifies a deadline of d time units. This expresses that the associated operation should be completed within d time units. The primitive $e < dl$ specifies a deadline for the event e that has to trigger within dl time units.

Table 1: Modelling

DSML	Time	Probability	Environment	API
Robochart	Y	Y	Y	Y
RobotML	N	N	Y	Y
GenoM	Y	N	N	Y
DSML for Adaptive System	N	N	N	Y
Frob	Y	N	N	N
DSL for Deployment	N	N	N	Y
Roboflow	N	N	N	N
Lightrocks	N	N	Y	Y
RAFCON	N	N	N	Y

Similarly, UML statechart has been extended with probability to model uncertainties [63]. RoboChart adopts probability as an addition to its notations that enables using a P-node junction for expressing probability, with the weight of the probability attached to each transition.

RoboChart enables using API to access common robotic operations. These operations can be integrated into RoboChart models as part of their operations and events [81].

RoboChart intends to provide a model of environment and platform in its future development. This will enable a developer to specify various configurations and settings that might be difficult to set up in a real-world scenario. This feature is not available in the current version but is part of the planned features of RoboChart.

Reasoning RoboChart was developed with formal notations with good support for formal reasoning. Also, RoboChart provides an automated technique for generating formal semantics in CSP notations that facilitates automated `V&V` techniques and tools, such as FDR model checker. This facilitates using mathematical theorems to check livelock-freedom, deadlock-freedom, determinism and refinement, in addition to the facilities for establishing an existence or absence of specific interesting behaviour, as well as using other facilities of the FDR model checker.

RoboChart adopts Object-Oriented simulation that enables direct mapping of the RoboChart constructs to simulations. It is planned to facilitate simulation using external simulation tools such as Gazebo [82] and V-rep [83]. These simulation tools are equipped with enough Physics engines for various simulation scenarios [81].

System RoboChart is currently being developed independently of any underlying platform or language. Also, the DSML provides facilities for developing platform-

Table 2: Reasoning Techniques

DSML API	Formal Semantics	V&V	Simulation
Robochart	CSP	Y	Y
RobotML	N	N	Y
GenoM	Fiacre	Y	N
DSML for Adaptive System	Alloy	Y	N
Frob	Equations	N	N
DSL for Deployment	N	N	N
Roboflow	Inference rules	N	N
Lightrocks	N	N	N
RAFCON	N	N	N

independent models and formal semantics. In the case of architectural style, RoboChart organises system models using two levels of the major constructs: module and controller, which has the advantage of improving modularity and simplifies component reuse [13].

Tool Support RoboChart provides a supporting tool called RoboTool that is being developed as an Eclipse plugin, using EMF for implementing the metamodel of RoboChart, Xtext for the text editor and Sirus for the graphical editor. In the case of executable generation, RoboChart focuses on generating mathematical models that are suitable for both `V&V` and simulation. Accessibility for the tool and the language itself is provided free for both personal use and research through its website [13].

2.3.2. RobotML

RobotML is another interesting DSML found in the literature that has been developed specifically for robotics with the aim of reducing the complexity with the use of sound SE concepts in developing robotics applications. Model-driven engineering (MDE) and Component-based software engineering (CBSD) form the basic underlying concept of RobotML. The language has been developed on eight proposed principles that define a good DSML for robotics systems. The requirements are (1) ease of use, (2) a specification of component-based robotic architectures, (3) independent of architectural style, (4) multiple heterogeneous target platforms [4] (5) platform-independent, (6) reuse, (7) a smooth evolution of the language, (8) reasoning facilities. Although reasoning with respect to time and other non-functional requirements is included as

[4]Ability to run the system components in different platforms, including simulators.

Table 3: System

DSML	Platform Indep.	Behaviour	Architectural Style
Robochart	Y	FSM	Controller/Module
RobotML	Y	FSM	Packages/Mission
GenoM	Y	FSM	Module
DSML for Adaptive System	Y	Tabular	Configuration
Frob	Y	FSM	Independent
DSL for Deployment	Y	NA	TSM
Roboflow	N	Flow chart(graph)	Flow graph
Lightrocks	Y	UML/P Statechart	Task/Skill/Action
RAFCON	Y	FSM	Hierarchical

part of these requirements. However, there is no clear consideration in using formal mathematical techniques for precise reasoning about the behaviour of the system [84].

Modelling RobotML was implemented using a UML metamodel for modelling robotic applications. It has been mentioned that it is possible to include time in the modelling system, which can be used for analysis, and scheduling, but there is no detailed information for the verification of temporal specifications. Additionally, the DSML lacks constructs for modelling probability. On the positive side, the language provides good support for using API to connect components, libraries and other existing components, a good provision for reusing resources. Constructs for the modelling environment are provided through external simulation engines.

Reasoning RobotML does not facilitate using formal semantics in the reasoning aspect, which makes it difficult to use theorem provers for validation and verification. This is understandable because RobotML is not based on mathematical formalisation. RobotML uses a different approach for effective modelling using the concept of ontology to enable effective communication between components. Also, simulation is facilitated through external simulation tools such as MORSE [85] and CycabTK [84,86].

System RobotML generates platform-independent models that support deployment and conversion for other platforms. System behaviour is defined using a combination of precompiled libraries of components, state machines and algorithms. Support for architectural style is provided using packages to organise system components and action for describing a mission [84].

Table 4: Tool Support

DSML	IDE	Framework	Executable Logic	Proprietary
Robochart	Y	EMF	Y	N
RobotML	Y	EMF	Y	N
GenoM	Y	N	Y	N
DSML for Adaptive System	Y	EMF/Kermeta	N	N
Frob	N	N	N	N
DSL for Deployment	Y	EMF	N	N
Roboflow	Y	PR2 ROS	Y	N
Lightrocks	Y	EMF	Y	N
RAFCON	Y	GTK+	Y	N

Tool Support A graphical modelling tool is provided for supporting the development of RobotML models. The tool was developed as an Eclipse plugin. In addition to using MOKA as an extension that is provided for modelling the executable components, together with the support of Acceleo (an eclipse plugin) for generating executable code. The output generated code is suitable for both simulations and physical hardware systems. Also, both the modelling language and its supporting tool are available to use free from the project website [87].

A distinguishing feature of RobotML is the concept of using ontology for defining robotic components and their relationships to realise a correct behaviour for the targeted system specifications. This concept has been advocated as a good way of reusing expert knowledge; similar to the approach of using ontology in other computing fields for problem-solving and raising the level of abstraction. For example, ontology is commonly used to describe systems in the area of Semantic web [88] and Artificial Intelligent [89]. Although this concept facilitates a useful way of sharing domain knowledge, however, it has less impact on system V&V and improving formal specifications. This is because formalisation involves using mathematics for precise specifications and proofs, more especially in developing a complex system like robotics [84].

2.3.3. GenoM

Another suitable DSML for our intended research is GenoM [31] (Generator Of Modules) tool, a tool for specification and generation of software modules [26]. It has been developed as a framework that encapsulates algorithms into an independent unit of a software component. Each software unit provides a well-defined system functionality or service that is suitable for CBSD, in such a way that each component is either an independent system or embedded-component as part of another system, irrespective of the underlying middleware. The components communicate independently based on contract, an as such components may not necessarily be aware of the actual operation of one another, and it is possible that they are developed independently with different programming languages. This contract facilitates communication, scheduling and control. Also, the connections between components are established using Component Description Language (CDL) and Internal Data Structure (IDS). Besides, each GenoM model facilitates system development from combinations of codes from different programming languages, such as C/C++, Prolog and Tcl scripts [90].

Modelling GenoM provides constructs for specifying time budget and the deadline for scheduling, control and analysis. The provided constructs are suitable for analysing real-time systems, but no information is available for handling probability and modelling environmental. However, on the positive side, GenoM supports using API for communication among components for both generated components and precompiled library components. Each component has a standard description using a standard template that makes it flexible for using multiple programming languages in developing the internal structure of a component.

Reasoning GenoM generates formal semantics of its model into formal specification languages BIP and Fiacre, which is used for automatic V&V using their supporting model checking tools TINA and D-finder, respectively. It has been mentioned that the formal semantics are suitable for simulation [31]. However, as at the time of this report, there is no clear information available for simulation in the GenoM tool [31].

System Initially, GenoM was developed for generating PocoLibs middleware components, but the recent version GenoM3 is independent of any platform and programming language [90]. Each component is independent; it has a specification file that defines its internal data, function, task and services. The specification file is generated using the provided standard Interface Definition Language (IDL) of Object Management Group (OMG). The modules are generated based on their provided specifications rather than the low-level behaviour, which facilitates encapsulating and integrating functions into components rather than developing low-level behaviour. The architectural style uses a modular approach to represent system components in defining system specifications.

Tool Support GenonM has been developed with a tool that generates executable components of a system. The tool has been developed independent of any framework, which is freely accessible through its website [91]. At the time of this study, there is no official standard release, but the tool is in active development and has updated supporting documentation.

2.3.4. A DSML for Adaptation logic

This is a textual DSML for specifying adaptation logic that facilitates separating adaptation logic from main system functionality. The aim of the language is addressing the challenges associated with Dynamic Adaptive Systems (DAS). The approach of the language combines the advantages of two commonly popular approaches: rule-based and optimisation-based. This concept takes the advantages of these two approaches whilst eliminating their shortcomings. The combination enables choosing a suitable configuration for a particular context based on the combination of constraints [27].

Modelling This DSML specifically targets modelling adaptation logic, so there are no constructs for time, probability or API. However, the DSML generates a text-based model of an environment using a collection of context variables [27].

Reasoning The DSML generates formal semantics for the Alloy model checking tool, which facilitates using the constraint solver of Alloy in generating the possible valid configurations that are used for formal verification. Also, textual simulation is provided using the values of the context variables of the adaptation model that facilitates generating all the possible valid configurations and ranking according to systematic scores for the configurations [27].

System The DSML generates platform-specific adaptation logics from the models and simulations. It has been mentioned that the DSML is specifically aimed at modelling the adaptation logic; for this reason, it lacks facilities for modelling high-level behaviour of a system and its architecture [27].

Tool Support A set of tools are provided to support using the DSML, which includes text editor, textual simulation and validation tools. The tools are developed using EMF for modelling and validation of Dynamic Adaptive Systems (DAS). The tool uses the facilities of the Kermeta platform for both generating semantics and simulation. However, the tool lacks facilities for generating executable logic. Both supporting tools and the language are freely available on the project website [92].

2.3.5. Frob

Frob (Functional Robotics) is a high-level control modelling language for robotics, which is being developed based on the concept of functional programming (separation of what from how). This approach hides the details of low-level robots' operations, including time. A function is considered as the first citizen entity that represents continuous behaviour. Therefore function serves as the basic unit of robotics system operations. The language aimed at providing an environment for developing correct and clean robotics applications without implementation details, which facilitates system development in a way that is suitable for formal reasoning.

One clear advantages of Frob is quick prototyping because it provides a high level facilities for testing control algorithms. Other advantages include promoting component reuse and improving modularity. The artefacts of this language provide a top-level structure of writing code that resembles mathematical equations, which is relatively similar to using mathematical equations for modelling systems, as commonly used in engineering and other related fields. This makes it a useful tool for designing and analysing experimental robotics [18].

Modelling Frob models a system with functions that changes with the flow of time. This abstracts time from the models with no provision for explicit constructs for modelling time, as well as the absence of facilities for modelling probability. The use of functional approach provides well-defined interfaces for connecting components. However, the language lacks support for using external API to access operations except through third-party interfaces such as C++. At the time of this work, there is no clear information about facilities for modelling environment in the language [18].

Reasoning Frob provides suitable models for formal reasoning. However, at the time of this report, it lacks support for automated formal reasoning. Also, there is no available information for simulation [18].

System Frob is a platform-independent DSML that uses functional structure to encapsulate generic code, which facilitates developing components that are independent of any platform [93]. System behaviour is defined as a continuous function that changes with the flow of time. This enables the state of the system to be determined at any given time. The language was designed with no specific architectural style, but is flexible enough to adopt any style [18].

Tool Support Frob was developed as an embedded DSML in Haskell. It is implemented as Haskel's library. As at the time of this work it lacks tools support, and no IDE was found specifically developed for Frob. The language provides an ideal tool for developing a quick prototype of a system that is flexible for both modification and subsequent implementation in other high-level languages, such as C or C++.

2.3.6. A DSL for Deployment

This DSL [94] is a kind of DSML developed specifically for modelling software deployment. The aim of the DSL is to address the complexity of deploying applications at the level of system integration. Thus, the DSL provides a high-level specification that separates deployment from the main software development, which makes the deployment stage of software completely independent of the target hardware platform [94].

Modelling The language focuses on modelling deployment for integration at a high level. However, the DSL lacks support for modelling low-level behaviour including time, probability, environment and API [94].

Reasoning The language does not facilitate using formal techniques for automatic verification. However, there are facilities for using constraints satisfiability checking for verifying the correctness of a model which has tool support for verification. Also, it lacks the simulation facilities.

System This DSL was developed independently of any platform with the aim of improving high-level integration. Therefore, it has no provision for modelling high-level behaviour. Similarly, it is independent of any architectural style but enables generating a specific application for a specific architecture using a feature-oriented approach [94].

Tool Support A graphical IDE tool is provided to support using the DSL that is based on EMF with Xtext [95] and Sirius [96] for the textual components. The supporting tool lacks facilities for generating executable logic, but it creates deployment models. At the time of this work, no information is available for accessing the language and its associated tools [94].

2.3.7. RoboFlow

The RoboFlow is a DSML for modelling and programming tasks, particularly for mobile manipulators. RoboFlow is specifically developed for a robot that operates in a human environment to provide support in carrying out a specific task, such as moving a load to another place or feeding disabled people. Visual flow graph programming is the central concept of the DSML, which uses boxes for defining procedures, similar to the popular approach of designing flowcharts. The procedures are associated with pre- and post-conditions that can easily be arranged in a flow graph to define a task. Each procedure consists of collections of operations and decisions, graphically represented with rectangles and diamonds, respectively. These provide constructs that facilitate easy programming for end-users with little low-level programming skills. The procedures are designed for a specific type of robotic platform which provides an easy way of interacting with specific robotic hardware. Overall this DSL provides simple graphical tools and programming language for modelling simple robotic applications for a specific platform [15].

Modelling This RoboFlow specifically focuses on task manipulation using predefined procedures available to targeted robot hardware. A user can arrange the predefined procedures in various configurations to describe a complete task. It was not developed with resources for modelling time, probability, environment or API. Also, the DSML is intended to be used for a specific system that operates in one particular environment within a narrow domain.

Reasoning The RoboFlow is equipped with the techniques of using logical inference rules for reasoning with a manual illustration. [15]. Also, this is the main facility for formal `V&V` in the DSML. The RoboFlow was explicitly developed for use in a specifically targeted hardware platform, as such it has no facilities for simulation [15].

System RoboFlow was developed for programming a specific hardware platform using the available predefined procedures with no means for modelling the low-level detail of a behaviour. The provided language uses simple procedures that do not consider the variation of architectural styles but just a flow graph [15].

Tool Support RoboFlow provides tool support for using the language with a graphical editor, which has been developed with Java applet. The language also uses the facilities of a package called PR2 for the demonstration aspect. Additionally, the facilities for modelling mobile behaviour was also provided with another package for autonomous navigation applications [97]. These two packages are part of the PR2 mobile manipulation platform developed for ROS [98]. The tool is available at this link [99].

2.3.8. LightRocks DSML

LightRocks (Light Weight Robot Coding for Skills) is a DSML for modelling task assembly that is based on the concept of assembling tasks and skills to describe system functionality. A task consists of collections of control flow that composes skills: a primitive operation. Each task is specific to a particular platform. The DSML is designed to provide an efficient separation of concern that relieves domain experts from the complexities associated with developing robotics applications. Various levels of abstractions are provided together with a supporting toolchain for improving communication between robotics and domain experts. An additional advantage of the approach includes promoting reuse of the existing components: skills and tasks. This has the advantage of effective modularity in developing robotics applications [100].

Modelling LightRocks focuses on assembling and composing skills and tasks to define behaviour. Therefore the DSML lacks constructs for modelling time, probability and environment. However, the DSML enables using API to access a precompiled library of skills [100].

Reasoning LightRocks has no support for using formal semantics and formal `V&V`. Similarly, there is no information about supporting simulation [100].

System LightRocks provides a platform-independent assembling process for robotics applications. However, the DSML does not provide constructs for modelling high-level behaviour, because it was developed to abstract low-level detail, that is suitable enough for composing skills and tasks to define behaviour. There is no available information about architectural style [100].

Tool Support The language is integrated into the MontiCore toolchain, which provides a modelling environment for using existing graphical and textual editors that are used for generating both code and executable logic from the models. Furthermore, the tool provides support for extending the code generation with other code generators for other languages and platforms. The tools are available as part of the MontiCore framework [101] with active support and documentation [100].

2.3.9. RAFCON

RAFCON (RMC advanced flow control) is a graphical tool for modelling task execution. RAFCON uses hierarchical state machines for visual programming to provide an easier way to develop robotic applications. The DSML is based on a Python interpreter that enables the use of Python resources, such as editors and compiler. To ease the debugging process, the DSML makes it flexible to start the state machine at any state to observe the behaviour of the system from any starting point. Additionally, the DSML provides debugging functionalities, error handling and mechanisms for recovery. The DSML supports the approach of divide and conquer to enable collaborative teamwork. This has the advantage of facilitating collaborative programming to develop big and complex applications [102]

Modelling The recent version of RAFCON lacks constructs for modelling time, probability and environment. However, it offers good support for API, which in turn provides good support for CBSD and enables the integration of modules developed with different languages [102].

Reasoning There is no clear information for supporting formal semantics that can be used for reasoning that is suitable for V&V as well as simulation.

System RAFCON is based on a Python interpreter; a platform-independent tool that uses a state-machine for modelling the behaviour of a system. In terms of architecture, the language supports hierarchical architectural style [102].

Tool Support RAFCON has a graphical editor for visual programming that supports using the DSML to ease prototyping as well as developing robotics applications. The supporting tool is provided as open-source through the project link [103].

In summary, most of the available DSMLs support and adopt graphical notations. Also, they promote the idea of generating an executable logic from the models. We find that state machine notation is the most commonly used structure for modelling behaviour, and EMF is the favoured framework for graphical modelling. The languages demonstrate various approaches to formalisation and architectural style. There is a common use of discrete approximation for handling continuous-time models. But only one DSML provides constructs for modelling probability. Interestingly, none of the DSMLs are proprietary, and none of the DSMLs provide facilities for generating test cases.

2.4. Formal Validation and Verification using DSML

This section provides a brief account of using formal techniques in V&V for the studied DSMLs, specifically for the development of robotic applications. Although many DSMLs have been developed that focus on enhancing the development processes, little attention has been paid to improving V&V and the use of formal techniques. RoboChart, GenoM and DSML for adaptive system are the only three DSMLs out of the nine studied DSMLs that used formal techniques for automatic V&V. A brief account of how these DSMLs use the formal method is highlighted in the subsequent paragraphs.

RoboChart uses CSP specifications for formal verification of systems. The formal semantics of RoboChart graphical models are automatically generated into CSP processes in a format that is suitable for automatic verification with the FDR model checker, which verifies system properties using refinement. On encountering failure, FDR generates a counterexample to illustrate where the system specification is not satisfied. Like most of the studied DSMLs, this approach is also limited to using a discrete-time approximation for both modelling and verification.

In the case of GenoM, formal verification was added as an extension to its original framework, for enhancing its capability with the use of formal techniques. In GenoM, there are two approaches to formalising, using either Fiacre/TINA toolbox or BIP. The first approach uses Fiacre/TINA toolbox, which provides automated techniques that have been developed in two transformation steps. The first step generates Fiacre formal specifications of the GenoM modules. The second step translates the generated Fiacre specifications into Timed Petri nets that provide suitable input for the TINA toolbox. The translation is mechanised using a Frac compiler [5], which performs syntax analysis, type checking and code optimisation while preserving the semantics the models. Then the TINA model checker is used for verifying temporal specifications of the modules [31].

The second approach for formalising GenoM was developed using a BIP framework, which has been implemented in three steps: (1) an automatic process for generating formal models of GenoM modules in the form of BIP models; followed by (2) adding constraints into the formal models; (3) D-Finder tool is used for the verification of the

[5]Frac is a compiler for Fiacre programming language.

generated formal models. The D-Finder tool is a verification tool developed for BIP, which provides facilities for identifying and eliminating deadlocks.

In the case of GenoM, the first major shortcoming of their approach is a state explosion. It has been reported that it takes more than two days (48 hours) to verify a combination of four modules using this procedure [104]. An effort for improving the performance has been reported [105] using real-time BIP framework, and also in using a multi-core version of real-time BIP. Secondly, understanding of BIP is needed for adding constraints in order to use the tool effectively. Additionally, according to the available information, it is easy to see that using BIP with GenoM is only capable of checking deadlock. On the temporal constraints, there is no clear information in the verification of temporal constraints using the continuous-time models.

It is good to note that, firstly, GenoM was initially developed with no basic foundation for using a formal method. The tool was developed for generating modules from a combination of models and code. As part of the effort of improving the tool, a technique was created for using templates to generate formal specification from module specification, rather than the actual implementation of the module. There is a concern here that at times the implementation may not reflect the specifications correctly. Secondly, a user must have an understanding of Fiacre modelling specification to be able to specify some properties for effective use of the GenoM tool for automated formal verification. Thirdly, in the case of encountering a failure, a counterexample is not generated in GenoM, which might make it difficult for the user to interpret the result. Additionally, these shortcomings have the potential to complicate further evolution of the tool. This is specifically important for formal verification of robotics applications because these robotics applications evolve rapidly to catch up with addressing future challenges.

As highlighted in Section 2.3.4, the DSML for adaptive systems uses Alloy formal specification language to verify interesting properties of the adaptation model. The DSML uses context variables to record all the factors that affect the correctness of the adaptation model, mostly factors that trigger an event, and also uses variants to record the possible options at the variation points. These context variables are used to generate the possible configurations of the adaptation model. Combinations of the context variables and variants are used to express the invariants and generate Alloy specifications. Then, Alloy constraint solving capability is used to solve the constraints and generate all possible valid configurations of the model. The tool ranks the generated configurations according to their properties and assigns a score value for each configuration. If the system encounters a failure, it indicates the configurations that did not satisfy the constraints [27].

There are limitations that made this DSML for adaptive systems unsuitable for using advanced formal verification techniques, more especially in the development of robotics applications. First, it has a narrow scope that focuses on modelling adaptation logic only, rather than for modelling a complete system. Additionally, most significantly, it is not a graphical language; it was entirely developed as a textual language with a textual model of the environment, including textual simulation.

Two other uses of formal semantics were highlighted in RoboFlow and Frob. For

RoboFlow, a manual procedure has been illustrated for using inference rules to generate formal semantics of RoboFlow models [15]. But the problem is that a manual procedure is not efficient and not suitable for V&V of large and complex systems like robotics applications. In the case of Frob, it has been mentioned that Frob is suitable for formal reasoning [106], but no detailed information is provided for a mechanised approach of using the formal semantics in V&V.

The success of using the formal method in industries has been reported with significant impacts, across various types of projects, such as automotive, business and critical systems. The report is a continuous survey that investigates the adoption of the formal method over the years [29]. One of the notable advantages of using formal techniques is quality improvement, which has been reported as one of the significant strengths of the formal method. Additionally, product quality plays a vital role in reducing both maintenance cost and effort. This is due to the capability of formal techniques of detecting bugs at the early development stages.

Most of the industries that tried formalisation in their projects have reported a good quality improvement of their product. They have also reported good motivation and interest to continue using formal techniques in their subsequent projects. But it has been reported that some kind of training or skills are needed for successful adoption of formal techniques. This can be supported with the development of automated tools [29].

The formal method provides a basic mathematical ground for automatic V&V of system behaviour. This capability of the formal approach in proving and establishing the correctness or absence of behaviour in a system makes it a suitable companion for improving DSML. Additionally, the formal approach facilitates a convenient way for the domain expert to verify their systems using formal techniques without the need for skills and understanding of the detailed concept of the Formal Method.

2.5. RoboChart in Detail

This section provides a discussion of RoboChart as the selected DSML for this work. This is because the semantics of RoboChart is generated in *tock-CSP*, which provides a good point for connecting refinement models and temporal logic. The section describes the constructs of RoboChart and provides examples to illustrate using the constructs to model a system.

RoboChart is an evolving modelling language developed with a subset of UML state machines with additional restrictions. It has been designed specifically for modelling and development of robotic systems to enhance the verification using advanced formal verification techniques that are based on formal mathematical models. The RoboChart modelling language provides a suitable construct for using formal techniques in the development of robotic systems, including constructs for modelling time and probability. This is not easy when using general programming languages. This approach enhances the verification aspect that has a high potential for improving the safety of autonomous robotic systems.

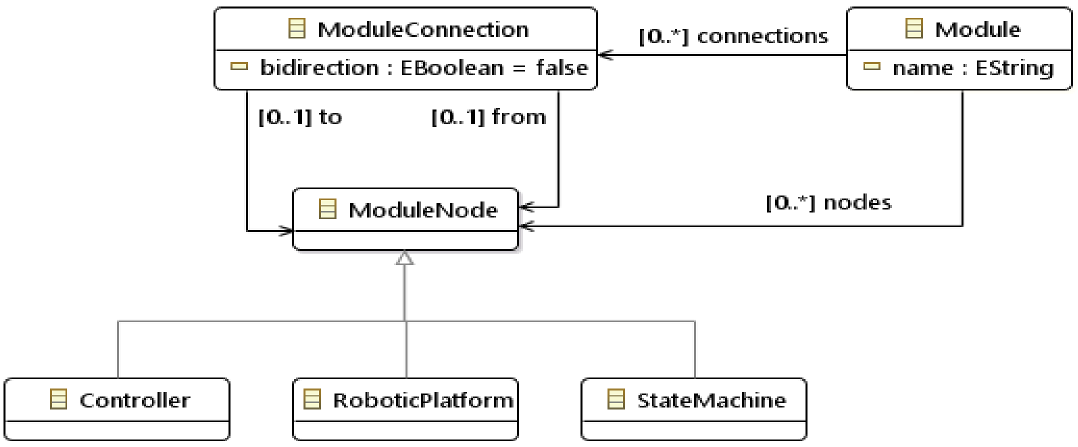


Figure 1: Metamodel of RoboChart models (source [1])

RoboChart Meta-model

The metamodel of RoboChart is presented in Figure **??** the high-level constructs and their relationships. The metamodel begins from a state machine at the bottom; a basic entity of RoboChart. The state machine has been proven suitable for modelling system behaviour. System behaviours are defined using the concept of the event-guard-action rule [81], in which a system reacts to its environment using an event, and takes appropriate action to perform an operation that triggers a change of state. This concept is captured with a state machine.

The metamodel of state machine is shown in Figure 2. In RoboChart, a state machine has a chain of connected states, which is either a simple state or composite state. A composite state contains other nested states and transitions. Each state has a maximum of three phases: entry, during and exit; representing an entering stage, an executing stage and an exiting stage, respectively. The states are connected with transitions, with an optional triggering event, a guard condition of type Boolean and an action. The initial state and final state describe the entry and terminating point of a state machine (or composite state). Additionally, a transition connects a state to a junction or connects a junction to another junction. A state node describes a stable state of a system, and an unstable state is described as a junction node, which models a decision point in a system.

As shown in Figure 1, the state machines are connected either in serial or in parallel to define a unit of control, called controller in RoboChart. A controller models a processing unit that describes an independent behaviour of a system. A group of controllers are connected to generate compound behaviour. Also, each controller can be connected directly to the hardware platform, either independently or in a group with other controllers.

To improve modularity, RoboChart provides an additional construct for organising and connecting related controllers together with their associated platform into a unit

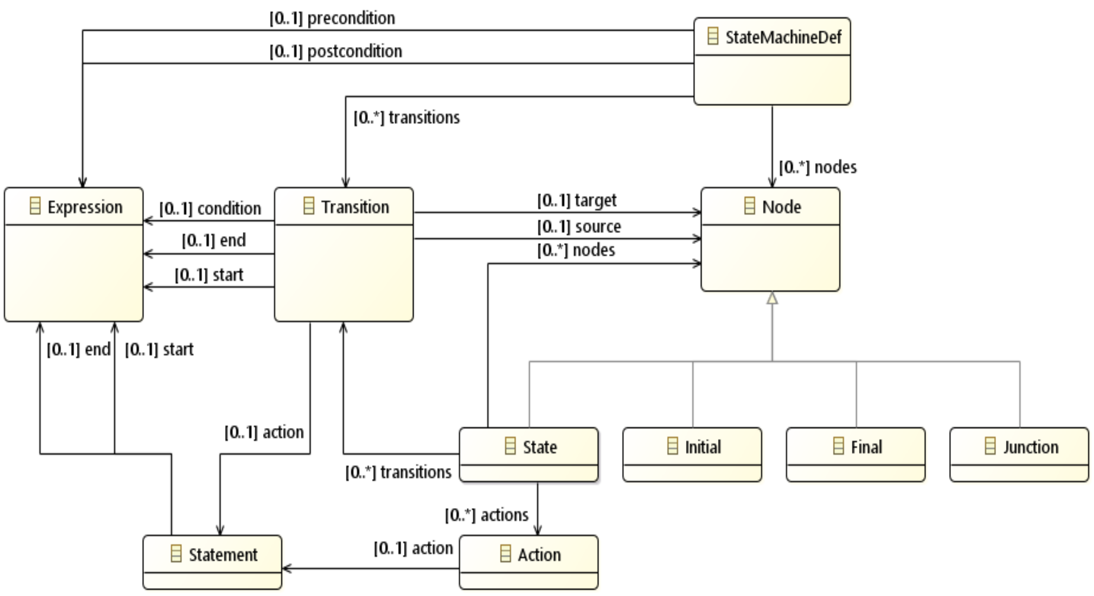


Figure 2: Metamodel of RoboChart state machine (source [1])

called a module, as shown in Figure 1. A module structures and organises system components in a well-defined entity that define system architecture. A system can have many modules, and one of the modules must define the system platform.

A robotic platform has variables, operations and events that describe the system properties, platform functionalities and the available facilities for interacting with the environment. The components are accessible to all the controllers and state machines inside the module.

RoboChart provides an additional top-level structure named package, which stores information of all the components used in modelling a system. A robot system has one or more packages; one of the packages must define the system module. The system module must contain the definition of the platform, and possibly other controllers.

In RoboChart, a communication flow is denoted with a connection arrow that connects two nodes: triggering and responding nodes. By default, communication is synchronous, which is the default communication between state machines, while asynchronous communication is marked with the label 'async'. The asynchronous communication is available between controllers, which communicate either synchronously or asynchronously. RoboChart can connect two events that have different identifiers. So, a triggering event may receive a response with an event that has a different identifier.

An event is the main interaction point between components, which facilitates communication between system components, such as controller, platform, and also the operating environment. There are two types of events: typed and untyped. A typed event communicates value during an interaction, while an untyped event facilitates interaction between components without communicating values.

At times there is a need to abstract the structure of a component that will enable the use of the provided component without its detailed implementation. RoboChart provides another construct *interface* for accessing components that are provided for common use. Thus, in RoboChart, the construct *interface* describes an expected component with its communication interface and its parameters, that include variables, operations and events.

Function and operation are also provided as part of the RoboChart constructs. A function evaluates a mathematical expression and returns a value, which is used to evaluate expressions or make subsequent decisions; for example, the function $add(x, y)$ returns the sum of x and y. Meanwhile an *operation* performs an action without returning any value. Typical examples of operations are move, stop and open.

RoboChart uses operations to describe a system functionality for carrying out a task. An operation is modelled using a state machine. An operation is associated with optional assertions using pre- and post-conditions. The use of assertions ensures the fulfilment of a contract; when an operation is invoked correctly with correct parameters and values the operation completes the contract correctly, which will ensure the correct behaviour of a component and system in general. For example, the operation $move(v, t)$ takes two parameters speed and time. An example of a pre-condition ensures that both parameters are positive ($v > 0$ *and* $t > 0$).

Each of the RoboChart components: platform, state machine and controller have their constructs such as variables, events, functions and operations, that describe the component. The constructs declared inside a component, are local to that component. A shared element is declared outside the component but has to be inside the enclosing component that contains all the components that share the element. For example, variables declared inside a state machine are only accessible to that state machine. In contrast, variables declared inside a controller are accessible to all the state machines inside the controller.

Apart from defining operations with state machines, RoboChart provides access to predefined operations from relevant packages using API. The packages contain standard operations, events and variables that are used for common system functionalities that are invoked with a simple API. The functionalities depend on the type of hardware platform. For example, a mobile platform can have functionalities such as start, move, turn; while a flying robot can have functionalities such as move, up, and down.

RoboChart constructs have been extended to include notations for expressing and modelling time constraints and other temporal properties. This is inspired by the constructs of Timed CSP [37] and CircusTime [38]. RoboChart is based on the concept of event-guard-action; a concept in which transition happens by triggering an event or satisfying the guard condition or both that leads to a change of state from one state to another. The condition for triggering an event or transition guard is expressed using time primitives, which is usually by specifying either a deadline or time budget.

Modelling Time in RoboChart

In the design of RoboChart, by default, each operation takes a negligible amount of time, approximately zero, that is represented with no time. An operation that takes time is accompanied by a time budget that is specified by time primitives such as $wait(t)$. The primitive $wait(t)$ allocates t units of time to the operation. Similarly, events have no time restrictions (deadline) unless it is explicitly specified with timing constructs. For example, an expression $e < \{t\}$ specifies that the event e must trigger before t units of time. The timed constructs are used to specify a budgeted time within which a particular event or action must happen.

Furthermore, clock primitives are also provided for specifying time elapse from a particular incident or event. For example, the primitives $since(S)$ specifies a time elapsed since the last clock reset. A clock reset is expressed using a primitive #C. Finally, the primitive $sinceEntry(S)$ specifies a time elapsed since entering a particular state S. Additional details of the provided timing constructs are available in the RoboChart documentation [13].

There are two forms of primitive expressions: simple and compound primitive. A simple primitive consists of one primitive expression, while a compound primitive combines multiple primitives using a logical operator, such as conjunction and disjunction. For example, with a clock variable t, we can write an expression $t > 5$ and $t < 10$ that specifies the time between 5 and 10 units of time. Also, RoboChart facilitates using time primitive with constant expression to specify a condition. This is similar to the concept of Timed Automata [107], which is limited to comparing one single time clock with a constant expression, using synchronous continuous-time clock [2]. For example, $t > 5$ is a valid expression but $t1 > t2$ is an invalid expression.

Modelling Probability in RoboChart

As highlighted in Chapter 2, RoboChart is the only DSML among the studied DSMLs that provides a construct for modelling probability; inspired by the concept of probabilistic UML statecharts [63]. This is an extension of UML with the added capability of using probability to model uncertainties and randomness. The provided construct for probability models the behaviour of a system in specifying uncertainties. The decision depends on both the state of the system and the state of its operating environment at that moment in time.

This concept of probabilistic UML arises from probabilistic automata [2]. In this concept, several edges emerge from a single node, with a probability attached to each edge expressing the likelihood of taking the transition. In a probabilistic UML statechart, this is represented using P-node [63]. RoboChart adopts a similar concept by labelling each of the P-node transitions with an expression $p(t)$. The expression quantifies the weight of taking any of the transitions. For example, Figure 3 illustrates modelling probability in RoboChart. The figure shows a portion of a system at a decision point with three equal options which are captured with a P-node junction that has three transitions, each associated with a probability of 1/3 .

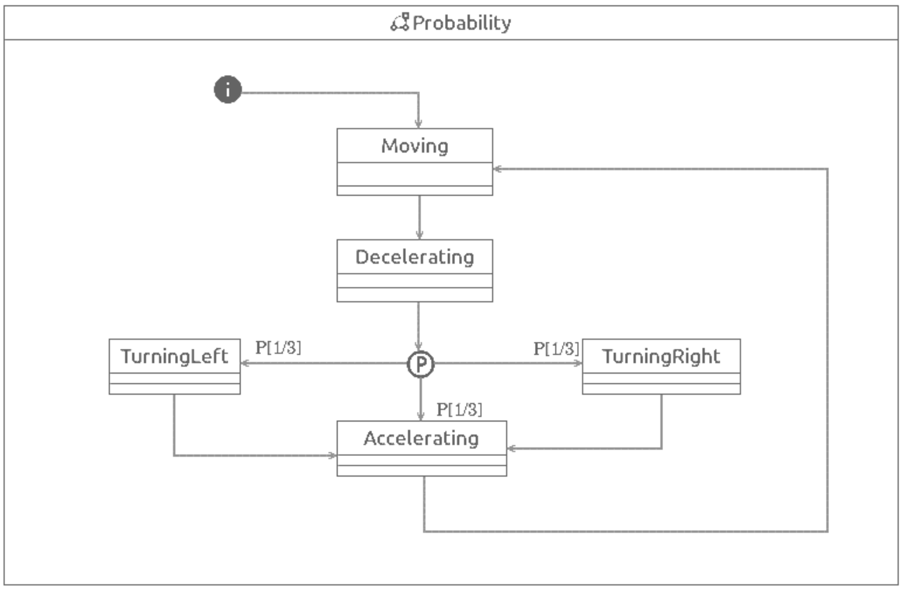


Figure 3: Modelling Probability in RoboChart

Example: Autonomous Navigation System (ANS)

Here we developed an example of a system ANS that moves from one position to another specific position. A relatively similar navigation system has been used in the literature, particularly in presenting two DSMLs, the RobotML [17] and the GenoM [105]. Here, we model the system differently from the way it was presented in the literature [17, 105], to suit the purpose of our work and illustrations. ANS is a navigation system that receives a target destination from the robotic platform. Then, the system determines its current position and computes a route to the destination; and follows the route to the destination while observing and avoiding obstacles. The system automatically stops on reaching the target destination.

The RoboChart module of this system is presented in Figure 4. The module consists of an automobile platform connected to four controllers, Navigator, Localiser, Observer and Trajectory. The function of the controller is as follows. The controller *Navigator* controls the movement of the hardware platform. The controller *Observer* observes the environment for detecting an obstacle on the path. The controller *Localiser* tracks the current location of the hardware in the operating environment. The controller *Trajectory* determines a route for navigating the robot to its destination.

The robotic platform has one basic operation *move*, that takes two parameters: direction to turn and distance to move. The operation *move* moves the platform in a specified direction. The system has three sensors for interacting with the environment: a camera for capturing the current state of the operating environment, an odometer for tracking the physical displacement of the system, and a GPS for positioning. The

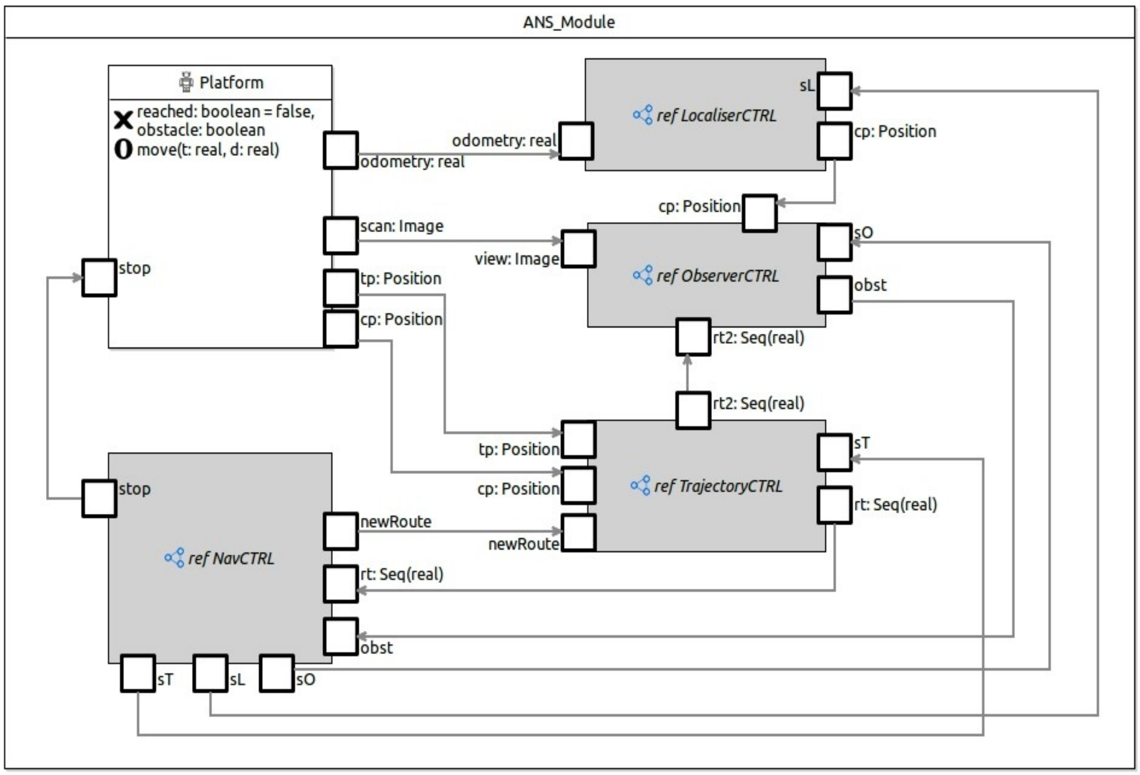


Figure 4: A RoboChart Module diagram for Autonomous Navigation System

GPS is used to set the target position and also computes the current position (as the initial position of the system). The GPS communicates both the target position and the current position using the typed events tp and cp, respectively. The controller *Trajectory* responds to these two events. Also, the platform sends the odometer readings to the controller *Localiser*, which handles the event *odometry*, and also sends the camera image to the controller *Observer* using the event *scan*. The platform responds to one event *stop* for stopping the mobile platform using a break.

The controller *Navigator* is presented in Figure 4, which responds to two events: *vrt* of type *Seq*(*real*) that describes the path to the destination, and the event *obst* that signals detection of an obstacle. The type *Seq*(*real*) is a sequence of pairs of data: move-turn, move-turn ... until the destination is reached. Each pair of move-turn specifies the distance to move and the direction to turn. The turning direction is a simple direction right and left. But this can be customised with more precise value, within the range of complete rotation $0 - 360$ degrees. The controller *Navigator* raises two events that are used to navigate the system: *newRoute* and *stop*. When the system encounters an obstacle, it uses the event *stop* to stop the platform and triggers the event *newRoute* to request a new route from the *Trajectory*.

The internal structure of the controller *Navigator* is presented in Figure 5. The

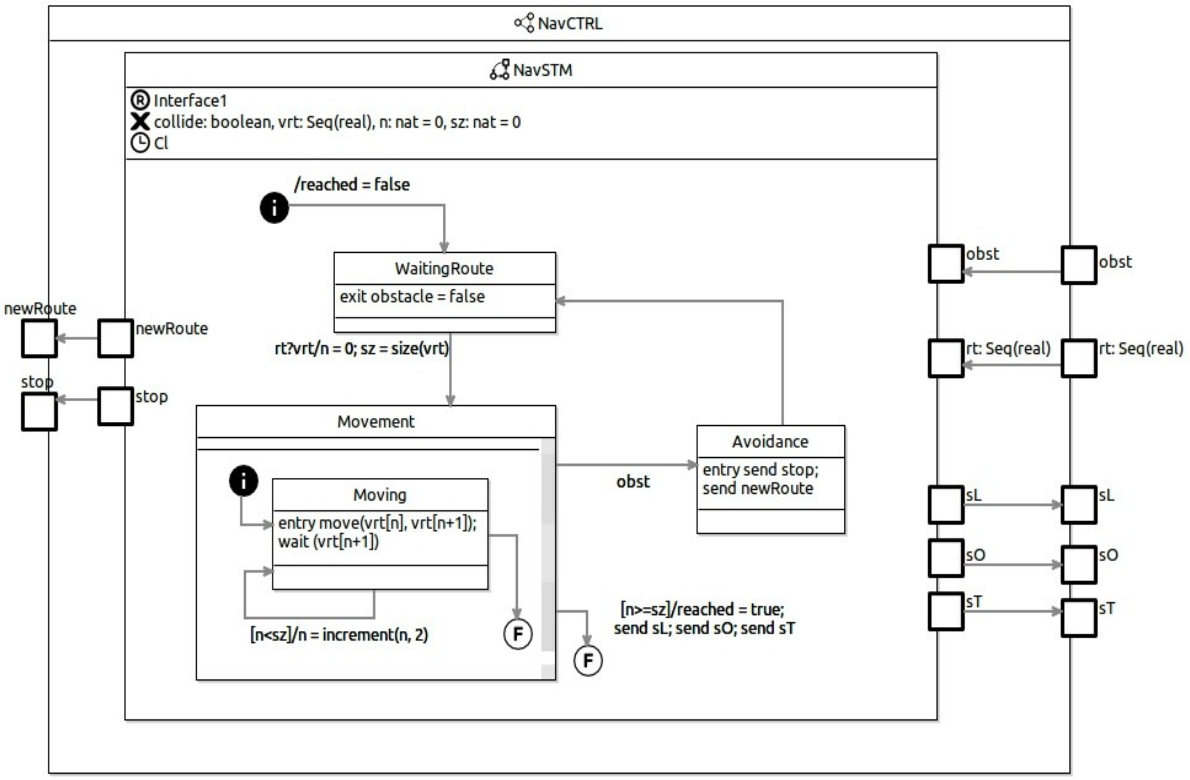


Figure 5: The Navigator Controller

structure is modelled with a state machine that begins in the state *WaitingRoute*. On receiving route information through the list *vrt*, it determines the length of the route using the function *size*(*vrt*) and stores the result in a local variable *sz*. The function *size*(*Seq*) is a simple function that takes a parameter of type sequence and returns the length of the sequence. The return value of the function *n* of type natural is the length of the sequence. The system reaches its destination when it reaches the end of the sequence in the variable *vrt*.

When the system encounters an obstacle with the occurrence of the event *obst*, the state machine moves to the state of *avoidance*, subsequently triggers the event *stop*, then requests a new path by triggering the event *newRoute*. The state machine waits for a new route at the state *WaitingRoute*. On receiving a new route, the system follows the route to the target destination. Also, the time primitives *wait*(2) in the state machine specifies a budget of 2 units of time for the operation *checkScan*().

The controller *Observer* responds to three events: *rt*2, *scan* and *cp*; and it raises one event *obst*. The event *rt*2 of type sequence specifies the path; *scan* of type image that captures the current state of the operating environment; particularly for detecting an obstacle, and the current position of the system *cp* of type position. The controller raises one event *obst* on detecting an obstacle. This is illustrated in Figure 4. The

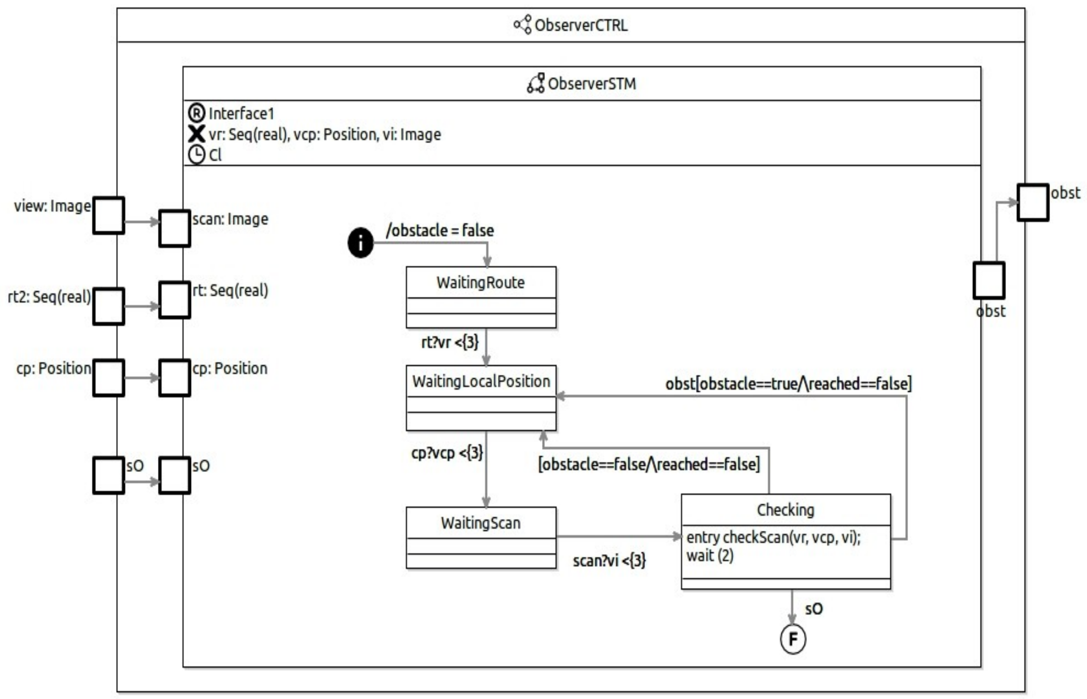


Figure 6: The Observer Controller

controller has one state machine as presented in Figure 6.

The internal structure of the controller *Observer* is modelled with a state machine in Figure 6. The state machine specifies a procedure for monitoring the state of the operating environment to detect an obstacle. The state machine begins at the state *WaitingRoute*, on receiving path information through the event *rt* it stores the path information in the variable *vr* and moves on to the state *WaitingLocalPosition*. On receiving the current position through the event *cp*, the system proceeds to the state *WaitingScan* and waits for the snapshot of the operating environment.

The operation *checkScan*() detects an obstacle. The operation takes three parameters, route, current local position and scan. The controller *Observer* uses this operation to raise the event *obst* for detecting an obstacle, then terminates at the final state receiving the signal on the event *sO*, which indicates that the system has arrived at its targeted destination, and therefore terminates the controller.

Similarly, the controller *Localiser* is shown in Figure 4 and its detail in Figure 7. The controller responds to one event *odometer* of type real, which communicates the readings of the odometer. Also, the controller raises two events that communicate the local position to two controllers: *Observer* and *Trajectory*. *Localiser* has one state machine that has two states: *Waiting* and *Localisation*. The state *Waiting* listens to the event *odometry* , while the state *Localisation* computes the local position using the operation *localisation*(), which takes one parameter *odometer*, for the odometry readings of the current local position of the robotic platform, and communicates the

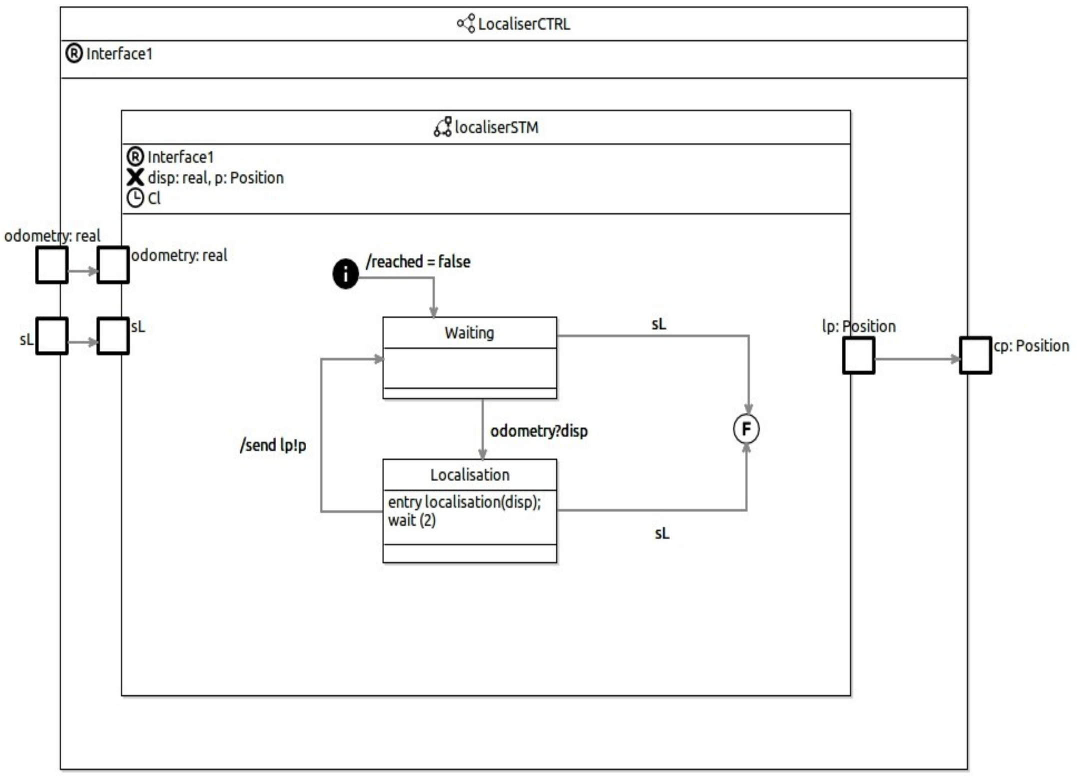


Figure 7: The Localiser Controller

value through the event *cp* to the controller *Observer*. The state machine waits for the next odometry data at the second state *Waiting*, or terminates when the variable *reached* becomes true, which specifies that the system has arrived at the destination.

Finally, the controller *Trajectory*, illustrated in Figure 4 with additional details in Figure 8, responds to three events, *tp*, *cp* and *newRoute*, for the target position, local position and new path, respectively. The controller raises two events $r1$ and $r2$ both of type route, that sends a path definition to the controllers *Observer* and *Navigator*.

The controller *Trajectory* has a state machine that begins at the state *Waiting*, and on receiving the two events *tp and cp*, it invokes the operation $findRoute()$ to compute a path to the destination. The detail of the internal structure of this operation is abstracted, which is either an implementation of an algorithm (software component) or external services using API [6]. In the end, the state machine sends the return value of this operation and terminates at the final state.

An additional detailed discussion of the timing constructs is available in Section 2.5, which discusses the detailed explanation of the timed primitives that we used in this example, such as $cp?vcp < 3$ and primitives $wait()$.

[6] Additional details of robotics API is available in this documentation `wiki.roboticsapi.org/Robotics_API_Development_Platform`

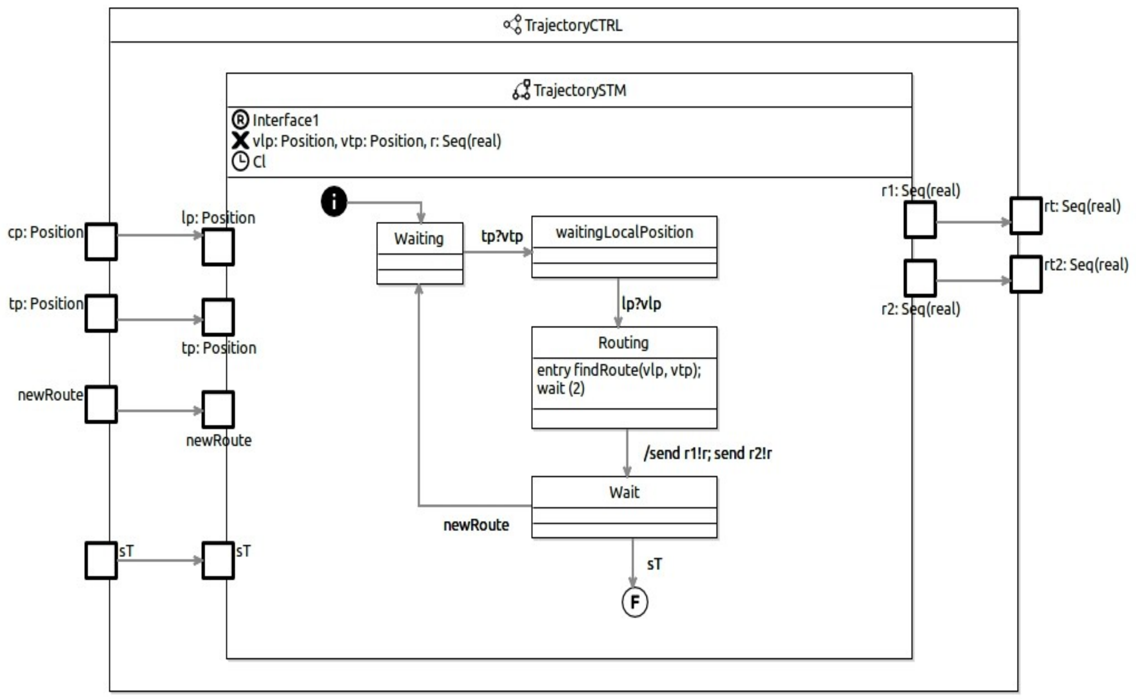


Figure 8: The Trajectory Controller

In summary, this section describes the structure of RoboChart and its untimed constructs, then follows with a demonstration of the untimed constructs using an autonomous navigation system as an example, which illustrates the use of the construct to model a system. The next section discusses the semantics of these constructs and their use for formal verification techniques.

Untimed Semantics

This section describes the formal semantics of RoboChart using CSP notation, and discusses the relevant semantics of the notations of RoboChart. A more elaborative discussion of capturing semantics using CSP notations is available in the literature [6]. There are two levels of details for describing semantics; the first untimed construct does not involve temporal specifications. The second approach does include temporal specifications, where the timing is essential for the semantics. This section focuses on the semantics of untimed behaviour using untimed CSP notations. The subsequent section discusses modelling the timed behaviour using the provided constructs of RoboChart.

The semantics of RoboChart is based on the framework Unifying Theory of Programming (UTP) due to its capability of expressing timing and probability. The aim of UTP is to develop a common ground for programming languages from the semantics perspective [108], which provides a common platform for using the semantics to

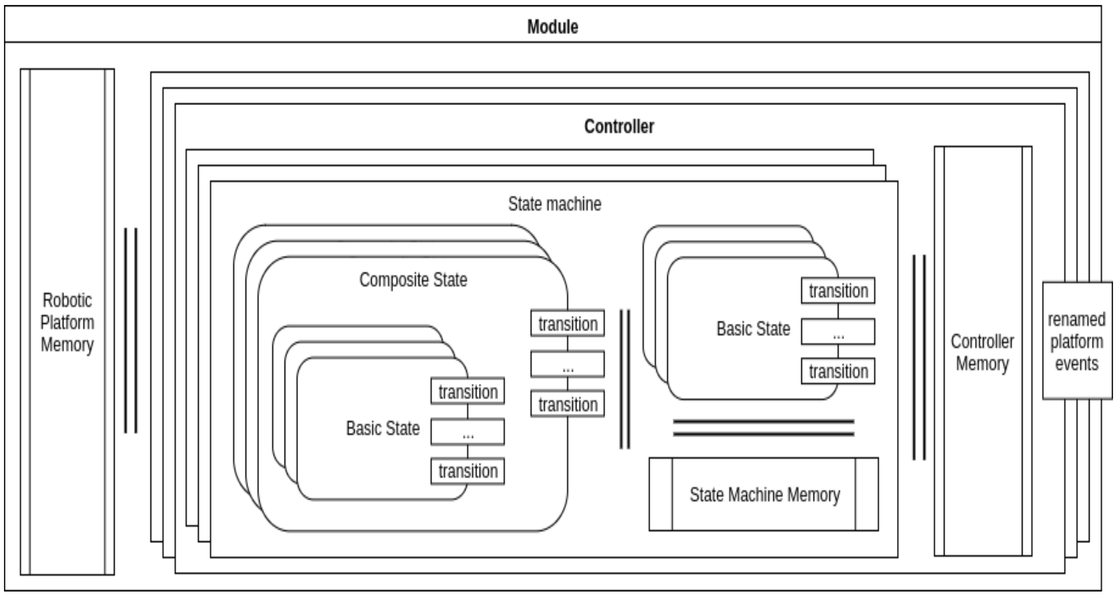


Figure 9: Stack structure of the semantics of RoboChart application, source [2]

compare and analyse different programming languages for a variety of paradigms. Therefore, UTP provides a suitable base for RoboChart semantics. The models of RoboChart have semantics in the form of denotational semantics of CSP in terms of UTP framework. As mentioned in the [37], the denotational semantics is the natural approach that matches the structure of CSP as well as its semantics. Furthermore, it is the same denotational approach used for analysis and reasoning about CSP models.

The semantics of RoboChart follows the hierarchical structure of its Metamodel. RoboChart modules consist of controllers that in turn contain a state machine, which describes the activities of the modules. Similarly, the semantics of a module is defined by composing the semantics of its controllers according to their interactions. Also, the semantics of a controller is deduced from the state machines that model the processes. This provides a hierarchical structure, as illustrated in Figure 9.

In Figure 9, each of the boxes represents a process. The box with double lines on both sides represents a memory process. The box with rounded corners represents a process that models a state. The process module (biggest box) models the whole system to capture the complete semantics of the system, which consists of the system memory (platform memory is the LHS box) in parallel with all the processes modelling the system controllers. Each one of the system controllers (bigger box) has its memory (controller memory is the RHS box). This is in parallel with the parallel compositions of all the state machines in the controller. At the bottom of the stack, each of the state machines has a memory (State Machine Memory) in parallel with the process modelling the parallel compositions of the composite state (big box), basic state (small boxes) and substate inside the state of a state machine.

RoboChart focuses on the behaviour of a system, which is expressed using a state

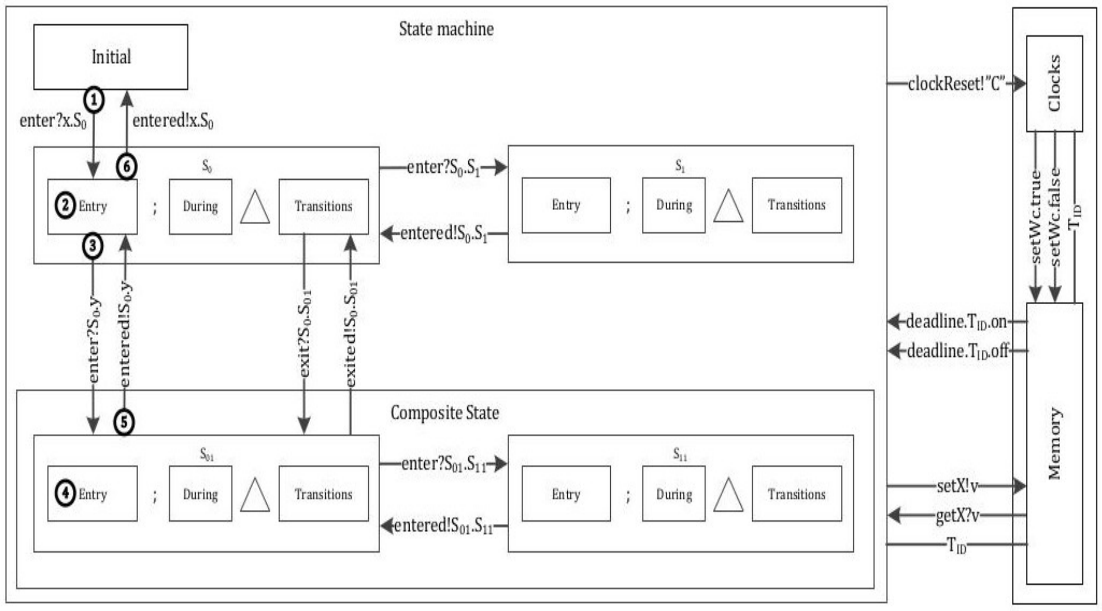


Figure 10: State Machine Semantics (source [2], fig 5).

machine that models system behaviour as a composition of the possible states of the system. Each state is modelled as an independent process. The composition of the state processes with the memory processes provides the semantics of the RoboChart state machine.

The detailed explanation of the semantics of RoboChart state machine is provided in [2], using an architecture that is outlined with a state machine in Figure 10. The figure consists of a state machine that has three states, two basic states: S1 and S2 and a composite state that also has two sub-states S11 and S12. Each state consists of three distinct parts: *entry* specifies the entry action, *during* specifies the actions between entry and exit, which may be interrupted with an outgoing *transitions*, and lastly, the *exit* specifies the exit actions. The *Initial* process models the initial transition from the initial state with the activation events *enter* and *entered*, which models a transition from one state to another.

For an overview of the examples that follow later, a typical structure of a RoboChart module consisting of controllers that contains state machines is illustrated in Figure 4 using the Autonomous Navigation System. In modelling the system, the only visible interactions are the robotic platform events, $stop$, $scan$, $odometer$, tp and lp. The events of a controller that is synchronised with the system platform are renamed to match the identifiers of the platform events. In the case of synchronous interactions and parallel compositions (|[ ]|) hiding (\)are used to synchronise the parallel controllers. An additional buffer is used in parallel with the controllers for asynchronous interactions. The semantics of these controllers together with the semantics of the platform memory (for the variables) produce the semantics of a module in RoboChart. Considering the previous example of ANS, the semantics of a RoboChart module is:

$$ANS\_Module = ANS\_Controllers[scan \backslash view]$$

The event *view* in the controller is renamed to match the corresponding platform name *scan*. The events connect to one another with the same identifier on the platform, so there is no need for the user to rename the events to match the platform.

Where the parallel controllers are structured as follow:

$$\begin{aligned} ANS\_Controllers = {} & ((NavigatorCTRL|[\{|newRoute, rt|\}]|TrajectoryCTRL) \\ & \backslash(newRoute, rt))|[\{|rt2, obst|\}]|ObserverCTRL|[\{|cp|\}]|Localiser \end{aligned}$$

The controller *navigator* synchronises with trajectory on the two events: *newRoute* and *rt*. The two controllers *navigator* and *trajectory* synchronise with the controller *observer* on the events *obst* and *rt*2, respectively. The controller *observer* synchronises with the *localiser* on the event *cp*. Also, all the events have similar names in both ending nodes, so there is no need to rename the events in the semantics.

The internal semantics of a controller is deduced from the composition and interaction of its state machines and the processes that model the memory, which captures the variable(s) used in the controller. This follows a similar pattern with the definition of a module, except here we use state machines instead of controllers. In the case of multiple state machines in a controller, the semantics of the controller consists of the parallel composition of all the state machines [2].

$$Controllers = StateMachine$$

The formal model of a RoboChart state machine is the composition of all the states inside the state machine that synchronises with its memory on the combination of the variables and events used in the state machine [2]. This is illustrated in Figure 10, which consists of a state machine with three states; two basic states S_1 and S_2 and a composite state consisting of two substates S_11 and S_12. The semantics is illustrated with a formal model of a process *StateMachine* as follows.

$$StateMachine = ((STM|[vars]|Memory)\backslash I)[f]$$

This process *StateMachine* is a composition of the states in the process STM and the process *Memory* that synchronise on the set containing the events for accessing the memory variables. The process hides all internal events (I) and uses a function (f) to rename all internal transitions identifiers to match the external events of the hardware platform.

The semantics of the composition of the states in the state machine is as follows.

$$STM = (Init|[EntryExitChannels]|States)\backslash(\Sigma \backslash PlatformEvents)$$

This process for modelling the semantics of the state machine STM consists of the initial transition process (*init*) that synchronised with the composition of the states (*States*) on the interaction channels (*EntryExitChannels*) in the state machine. And the process hides all the events (Σ) from the environment using the operation $\setminus$ except the platform events.

The process *Init* specifies the initial state, which uses an entry channel *entered* to send a request for entering another state. The corresponding response from the target state is through the channel *entered* for confirmation of entering the state. The semantics is specified as follows:

$$Init = enter.requester.target \rightarrow entered.requester.target \rightarrow SKIP$$

A state machine begins with an *initial* state, then follows with remaining states in the state machine. The process *initial* describes an initial transition into the state machine. Transition to other states is described using the activation events *enter* and *entered*. Similarly *exit* and *exited* are events for deactivating a state. Each one of these four events has two parameters, a state that initiates the request and the target state. For example, in Figure 10, the process $enter?\,x.S_0$ models a request for activating state S_0 from a state x, and the subsequent event $entered!\,x.S_0$ models an acknowledgement of the request.

The process *State* describes the composition of states in the state machine [13]. In a RoboChart model, each state machine begins from the initial state specified in the *Init* process, and exiting is always through the final state, except in the case of instant termination where the state machine just terminates by raising a termination event.

In the case of the example ANS, the state machine in the navigator controller is presented in Figure 5. A transition for activating a sub-state *Moving* from the basic state has a formal model using a process *T_init*, as follows.

$$T\_init = enter.Movement.Moving \rightarrow entered.Movement.Moving \rightarrow SKIP$$

In the same manner, the formal model of a transition from one state to another state follows the same pattern. For instance, in the state machine navigator inside the navigator controller, a transition from the state *avoidance* to another state *WaitingRoute* has a formal semantics, illustrated as follows.

$$T = enter.Avoidance.WaitingRoute \rightarrow entered.Avoidance.WaitingRoute \rightarrow SKIP$$

Each state machine has a memory that contains all its variable(s). There are two types of provided channels, *setChannel* and *getChannel*, for accessing each variable in the memory. For example, the process *P_memory* has a formal semantics, illustrated using the operator of external choice as follows.

$$P\_memory(v) = (getChannel!\,v \rightarrow P\_memory(v) \;\square\; setChannel?\,vnew \rightarrow P\_memory(vnew) \square \ldots)$$

Additionally, the state machine navigator has a memory process *Pm* that models all its variables as follows.

$$
\begin{aligned}
Pm(vrt, n, sz \ldots) = {} & (get\_vrt!\, vrt \rightarrow Pm(vrt, n, sz \ldots)[]set\_vrt?\, new\_vrt \rightarrow \\
& Pm(new\_vrt, n, sz \ldots)[]get\_n!\, n \rightarrow Pm(vrt, n, sz \ldots)[] \\
& set\_n?\, new\_n \rightarrow Pm(vrt, new\_n, sz \ldots)[]get\_sz!\, sz \rightarrow \\
& Pm(vrt, n, sz \ldots)[]set\_sz?\, new\_sz \rightarrow Pm(vrt, new\_n, new\_sz \ldots)[] \ldots)
\end{aligned}
$$

A guarded transition is a transition that has guard expression. The semantics of a guarded transition includes the semantics of the variable used in expressing the guard. For example, in the state machine navigator, the transition from the state *WaitingRoute* to the state *Movement* has a formal semantics that is specified by the process *Tm* as follows:

$$
\begin{aligned}
Tm = {} & rt?\, vrt \rightarrow set\_vrt!\, vrt \rightarrow (set\_n!\, 0 \rightarrow SKIP; get\_vrt?\, vrt \rightarrow \\
& set\_sz!\, length(vrt) SKIP; enter.WaitingRoute.Movement \rightarrow \\
& entered.WaitingRoute.Movement \rightarrow SKIP)
\end{aligned}
$$

The transition from the state *WaitingRoute* to the state *Movement* is connected with an event *rt*, so the process *Tm* begins with that event *rt*. It then follows with the memory processes: *set_vrt*, *set_n*, *get_vrt* and *set_sz* for accessing the memory variables. Then the process continue with the events for both entry request *enter*, and the corresponding response *entered* for the transition from the state *WaitingRoute* to the state *Movement*.

The formal model of a state consists of the events inside the state machine for the three possible categories: *entry*, *during* and *exit*, followed by the choice of possible transitions that connect the state to other states. An example is the sub-state *Moving* in the state machine *navigator* in Figure 5, which calls the two operations *move* and *wait* for the amount of time specified in the primitives *wait*(), then takes one of the two possible transitions. Formal semantics of the untimed model is defined with a process *Pwr* as follows.

$$
Pwr = get\_vrt?\, vrt \rightarrow get\_n?\, n \rightarrow moveCall \rightarrow moveRet \rightarrow SKIP; STOP/\backslash(t1 \ and \ t2)
$$

The process *Pwr* accesses the variable using the memory processes $get\_vrt?\, vrt$ and $get\_n?\, n$, then calls the move operation using the two events *moveCall* and *moveReturn*, and finally waits for the availability of any of the two transitions *t*1 and *t*2.

In conclusion, this section briefly discusses the untimed semantics of the RoboChart model. The section illustrates a composition of the formal semantics of state machines that form the semantics of a controller. In the same way, the formal semantics of all the controllers are put together to form the semantics of the whole system behaviour. The following section discusses timed semantics of the RoboChart model.

Discrete Timed Semantics

This section discusses the semantics of the discrete timing constructs of RoboChart. We discuss using CSP and its extension *tock-CSP* in capturing system specification using discrete-time models. An example is provided that illustrates using the construct of RoboChart for modelling temporal specifications of a system.

The initial version of CSP notation has been as an untimed language, which has been extended with the notion of time that facilitates analysing real-time systems. Thus, *tock-CSP* is a timed version of CSP that has an additional independent timed event *tock*, which marks the passage of time. The event *tock* represents a unit of time like 'a second' or 'a tick' of a wall clock, but the symbol *tick* (✓) is already part of CSP, which representes a successful termination of a process, so the symbol *tock* is used for describing the progress of time, which also serves as a time synchronisation point for all the processes and their operating environment. Therefore, any two or more events that happen between two consecutive *tock* events, are considered as they occur simultaneously. Thus, using *tock* facilitates the analysis of timed systems without the need for learning new notations and utilise the existing tools of CSP. The timed semantics of RoboChart is generated using *tock-CSP*, which is used in RoboTool for the verification of RoboChart models with the support of the FDR model checker for automatic verification [37].

To demonstrate modelling the timed specifications in RoboChart, the previous example ANS (presented in Section 2.5) has deadlines for specifying time elapsed, which makes the system synchronise with its operating environment in real-time. The expression $scan?vi < \{3\}$ sets a deadline of three timed units for the event $scan$. Another example is the expression $lp?vlp < \{3\}$ that specifies three time units for the event lp. Similarly, another deadline is added to the event r for receiving a new route on time, to avoid stopping the platform in one spot while waiting for a new route.

In RoboChart, deducing the timed semantics of a particular state machine is the parallel composition of the processes of the state machine activities, the processes of the clock and the processes of the memory that synchronise on a set consisting of their common events; and is illustrated as follows.

$$(STM|[a \cup dc]|(Pmemory|[t \cup w]|Pclock)\backslash w)\backslash I))[f]$$

The processes state machine STM and $Pmemory$ come from Section 2.5. The other processes $Clock$ and $Memory$ synchronise on a set combining the events for setting the clock (w) and timed guarded transitions (t). They also synchronise with the state machine process STM on a set containing a combination of events for accessing the memory variable (a) and clock (dc) for resetting the clock after the deadline. Then, the process hides all internal events (I). Also, the function (f) renames the transitions identifiers to match the platform external events.

In deducing the semantics of RoboChart models, the semantics of the primitive $wait(t)$ is its corresponding CSP process $WAIT\ t$, inspired by the constructs of Timed CSP. Similarly, the semantics of a deadline for an action $A < \{t\}$ is deduced as $A \triangleright t$, inspired by Circus Time. But in the case of a deadline for triggering an event, it is only enforced after entering the source state that satisfies the guard condition attached to the transition. In this case, two events are provided, $deadline.T.on$, when the guard condition is satisfied, and the second event $deadline.T.off$ when the guard condition is not satisfied. These are composed with the process *during* for the source state [2]. For instance, consider a transition *Ta* that has a deadline *da* time units, the formal semantics of the process *Pa*, as follows:

$$Pa = deadline.Ta.on \rightarrow ((deadline.Ta.off \rightarrow SKIP) \triangleright da); Pa$$

This process *Pa* synchronises on the event *deadline.Ta.on* and then also synchronises on the event *deadline.Ta.off* within the specified time interval *da* time units, and then the process and behaves as the process *Pa* again.

In addition to the time primitives for specifying time budget, RoboChart has a time clock for specifying a time elapse. The semantics of the time clock is encoded using a combination of boolean variables and an auxiliary CSP process that synchronises with the memory processes [2]. For example, a transition between two states that has a guard $since(C) < db$, which is modelled with a process *Tb* is encoded as part of the process *Pmemory*, below. The guard expression $since(C) < db$ is encoded using a variable *mb* that sets and resets the channel *setmb* as either true or false.

$$Pmemory(\ldots, mb) = (\ldots, \ \Box\ setmb?\,mbnew \rightarrow memory(\ldots, mbnew) \ \Box\ mb \& Tb \\ \rightarrow memory(\ldots, mb)$$

The process *Pmemory* offers multiple channels for accessing the memory, including the channels for accessing clock variable *mb*. Here the process illustrates using the channel *setmb* to set the clock time to a new value *mbnew* then the process behaves as *Pmemory* with the new value.

In synchronising the memory processes, a process for the waiting condition (*wcp*) is introduced as follows.

$$P(Tb) = Tb \rightarrow P(Tb)$$

$$wcp = P(Tb) \triangle wcp\_reset$$

$$wcp\_reset = clockReset.C \rightarrow setmb!\,false \rightarrow wcp\_body$$

$$wcp\_body = (P(Tb) \triangle_{db} setmb!\,true \rightarrow P(Tb)) \triangle wcp\_reset$$

The process $P(Tb)$ is ready to perform the event *Tb*, which is interrupted by resetting the clock *wcp_reset* using the event *clockReset.C*. The event *wcp_reset* updates the availability of *Tb* with setting the value of the variable *mb* false and behaves as the process *wcp_body*, which offers the event *Tb* for the time *db* units of time and then reset the memory back to true, then continues to offer the event *Tb* again.

For example, to complete the timed semantics of the state machine navigator Figure 5 its semantics is express in the process *Tp*, as follows:

$$Tp = (NavSTM|[vc \cup cc]|(NavMmr|[gt \cup wc]|NavClock)\backslash w)\backslash it)[f]$$

In this process Tp, the memory of the state machine navigator $NavMmr$ synchronises with the process $NavClocks$ on the set consisting of the combination of the guarded transitions gt and waiting conditions wc. After that, the process Tp hides all the waiting conditions wc to hide the event from the operating environment, and then also synchronises with the process $NavSTM$ on a set that combines both the events for accessing the variable channels (vc) and the clock channels (cc), while hiding all the internal events in the set it and renaming the transitions to match external events of the hardware platform.

Similar to the semantics of the untimed model, the timed semantics of all the state machines in a controller are combined in parallel to form the timed semantics of a controller. In the same manner, the timed semantics of a system is the parallel combination of the timed semantics of its controllers.

In conclusion, this section discusses the semantics of RoboChart models using the CSP notations, as well as *tock-CSP* for timed semantics of RoboChart. This provides a way of analysing systems together with their temporal specifications. The tool RoboTool automates the procedure of generating the CSP semantics with the added event *tock* for capturing the progress of time. The FDR [109] provides support for automatic verification of the generated *tock-CSP*.

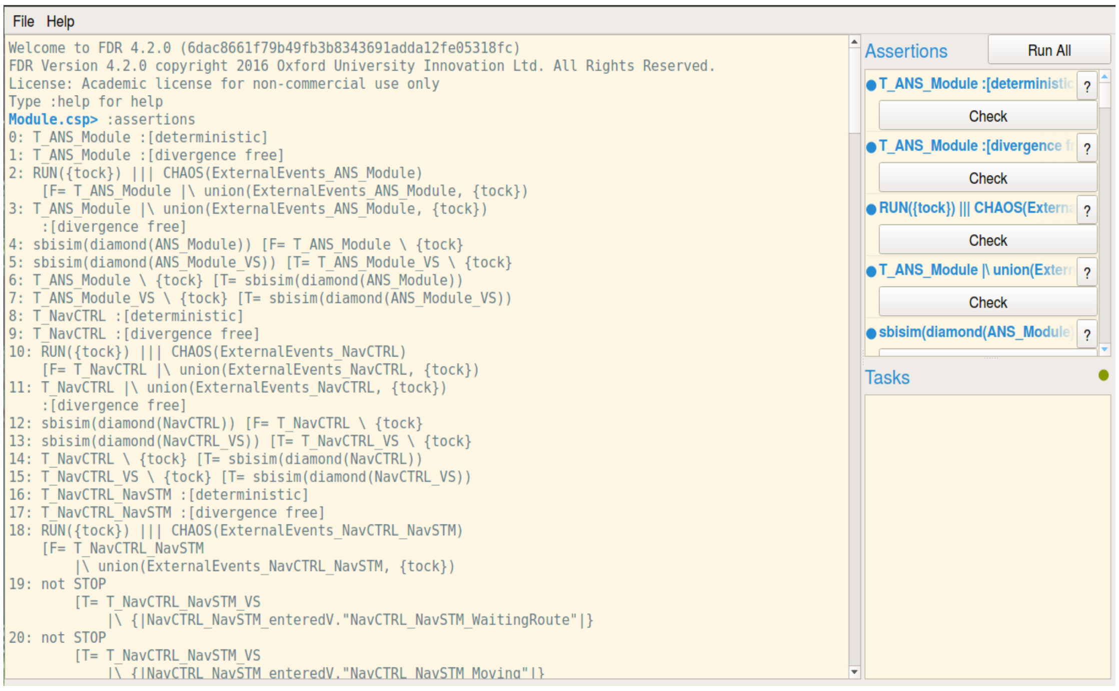
At the time of conducting this work, RoboChart was still evolving. As such we focus on *tock-CSP* that capture the generated semantics of the RoboChart models. In the next section, we are going to discuss using FDR in the verification of a *tock-CSP* model.

2.6. Automatic Verification

The supporting tool of RoboChart is RoboTool that generates CSP semantics of a RoboChart model from the graphical models. The generated CSP specifications are generated in a machine-readable format of CSP known as CSP_m [6]. RoboTool generates both untimed semantics and timed semantics as described in Section 2.5. RoboTool has been integrated with the FDR model checker [109] for both analysis and verification. Therefore, system properties specified using RoboChart are automatically translated into CSP, which is verified with the FDR model checker.

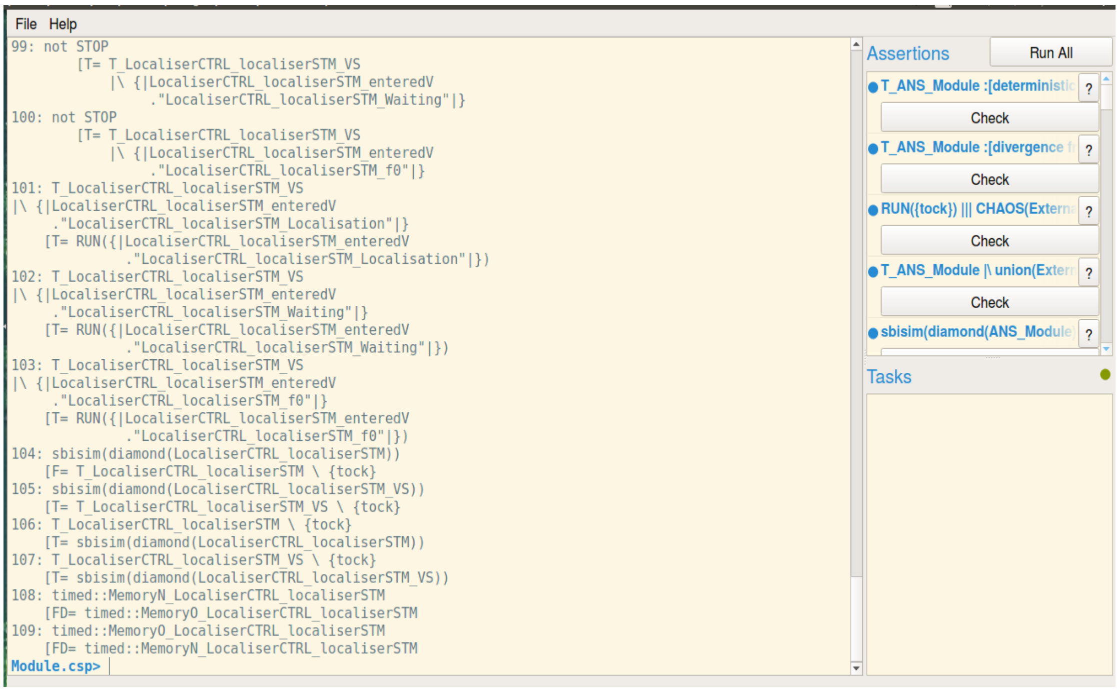
The FDR model checker is the primary tool for automated analysis and verification of CSP models. The FDR uses the concept of refinement to verify the requirements of a system. When FDR encounters a failure, where the requirement does not satisfy the system specification, it generates a counterexample to indicate where the requirement does not satisfy the specification. This facility is used to verify deadlock, determinism, divergence and other formulated requirements of the users. Animating system behaviour is also available using an animator Process Behaviour Explorer (ProBe), accompanying the FDR model checker. This makes it possible to visualise and explore the behaviour of individual processes that describe the behaviours of a system [6].

Integrating RoboTool with FDR involves loading the generated CSP specification (in .csp file) from the RoboChart package into FDR. Therefore, a system modelled in RoboTool is automatically verified for deadlock-freedom, determinism and divergence. Other possible verifications include using assertions to verify customised sys-


File
Help
Assertions
Run All
T_ANS_Module :[deterministic
?
Check
T_ANS_Module :[divergence f
?
Check
RUN({tock}) ||| CHAOS(Extern
?
Check
T_ANS_Module |\ union(Exter
?
Check
sbisim(diamond(ANS_Module
?
Tasks

```
Welcome to FDR 4.2.0 (6dac8661f79b49fb3b8343691adda12fe05318fc)
FDR Version 4.2.0 copyright 2016 Oxford University Innovation Ltd. All Rights Reserved.
License: Academic license for non-commercial use only
Type :help for help
Module.csp> :assertions
0: T_ANS_Module :[deterministic]
1: T_ANS_Module :[divergence free]
2: RUN({tock}) ||| CHAOS(ExternalEvents_ANS_Module)
    [F= T_ANS_Module |\ union(ExternalEvents_ANS_Module, {tock})
3: T_ANS_Module |\ union(ExternalEvents_ANS_Module, {tock})
    :[divergence free]
4: sbisim(diamond(ANS_Module)) [F= T_ANS_Module \ {tock}
5: sbisim(diamond(ANS_Module_VS)) [T= T_ANS_Module_VS \ {tock}
6: T_ANS_Module \ {tock} [T= sbisim(diamond(ANS_Module))
7: T_ANS_Module_VS \ {tock} [T= sbisim(diamond(ANS_Module_VS))
8: T_NavCTRL :[deterministic]
9: T_NavCTRL :[divergence free]
10: RUN({tock}) ||| CHAOS(ExternalEvents_NavCTRL)
    [F= T_NavCTRL |\ union(ExternalEvents_NavCTRL, {tock})
11: T_NavCTRL |\ union(ExternalEvents_NavCTRL, {tock})
    :[divergence free]
12: sbisim(diamond(NavCTRL)) [F= T_NavCTRL \ {tock}
13: sbisim(diamond(NavCTRL_VS)) [T= T_NavCTRL_VS \ {tock}
14: T_NavCTRL \ {tock} [T= sbisim(diamond(NavCTRL))
15: T_NavCTRL_VS \ {tock} [T= sbisim(diamond(NavCTRL_VS))
16: T_NavCTRL_NavSTM :[deterministic]
17: T_NavCTRL_NavSTM :[divergence free]
18: RUN({tock}) ||| CHAOS(ExternalEvents_NavCTRL_NavSTM)
    [F= T_NavCTRL_NavSTM
        |\ union(ExternalEvents_NavCTRL_NavSTM, {tock})
19: not STOP
        [T= T_NavCTRL_NavSTM_VS
            |\ {|NavCTRL_NavSTM_enteredV."NavCTRL_NavSTM_WaitingRoute"|}
20: not STOP
        [T= T_NavCTRL_NavSTM_VS
            |\ {|NavCTRL_NavSTM_enteredV."NavCTRL_NavSTM_Moving"|}
```

Figure 11: Sample assertions that can be verified using RoboTool and FDR for the ANS example (1 of 2)


File
Help
Assertions
Run All
T_ANS_Module :[deterministic
?
Check
T_ANS_Module :[divergence f
?
Check
RUN({tock}) ||| CHAOS(Extern
?
Check
T_ANS_Module |\ union(Exter
?
Check
sbisim(diamond(ANS_Module
?
Tasks

```
99: not STOP
        [T= T_LocaliserCTRL_localiserSTM_VS
            |\ {|LocaliserCTRL_localiserSTM_enteredV
                ."LocaliserCTRL_localiserSTM_Waiting"|}
100: not STOP
        [T= T_LocaliserCTRL_localiserSTM_VS
            |\ {|LocaliserCTRL_localiserSTM_enteredV
                ."LocaliserCTRL_localiserSTM_f0"|}
101: T_LocaliserCTRL_localiserSTM_VS
|\ {|LocaliserCTRL_localiserSTM_enteredV
     ."LocaliserCTRL_localiserSTM_Localisation"|}
    [T= RUN({|LocaliserCTRL_localiserSTM_enteredV
              ."LocaliserCTRL_localiserSTM_Localisation"|})
102: T_LocaliserCTRL_localiserSTM_VS
|\ {|LocaliserCTRL_localiserSTM_enteredV
     ."LocaliserCTRL_localiserSTM_Waiting"|}
    [T= RUN({|LocaliserCTRL_localiserSTM_enteredV
              ."LocaliserCTRL_localiserSTM_Waiting"|})
103: T_LocaliserCTRL_localiserSTM_VS
|\ {|LocaliserCTRL_localiserSTM_enteredV
     ."LocaliserCTRL_localiserSTM_f0"|}
    [T= RUN({|LocaliserCTRL_localiserSTM_enteredV
              ."LocaliserCTRL_localiserSTM_f0"|})
104: sbisim(diamond(LocaliserCTRL_localiserSTM))
    [F= T_LocaliserCTRL_localiserSTM \ {tock}
105: sbisim(diamond(LocaliserCTRL_localiserSTM_VS))
    [T= T_LocaliserCTRL_localiserSTM_VS \ {tock}
106: T_LocaliserCTRL_localiserSTM \ {tock}
    [T= sbisim(diamond(LocaliserCTRL_localiserSTM))
107: T_LocaliserCTRL_localiserSTM_VS \ {tock}
    [T= sbisim(diamond(LocaliserCTRL_localiserSTM_VS))
108: timed::MemoryN_LocaliserCTRL_localiserSTM
    [FD= timed::Memory0_LocaliserCTRL_localiserSTM
109: timed::Memory0_LocaliserCTRL_localiserSTM
    [FD= timed::MemoryN_LocaliserCTRL_localiserSTM
Module.csp>
```

Figure 12: Sample assertions that can be verified using RoboTool and FDR for the ANS example (2 of 2)

tem behaviour and also enable the verification of the expected hardware specifications as well as reachability analysis. Additionally, RoboTool automatically generates multiple assertions for verifying the correctness of a system [81].

For example, in the previous example ANS, the state machine navigator from Figure 5, an assertion to verify that the navigator state machine is deterministic is as follows.

- Checking deterministic behaviour
  ```
  Assert NavSTM :[deterministic]
  ```
- Checking divergence behaviour
  ```
  assert NavSTM :[divergence-free]
  ```
- Checking time locks
  ```
  assert (RUN({tock})|||CHAOS(EE_NavSTM))
            [F= (NavSTM \ union(EE_NavSTM, {tock}))
  ```

The assertion verifies that the LHS process $RUN()$ always offers the event *tock* concurrently (`|||`) with another process $CHAOS()$, which performs any of the external events of the state machine navigator (EE_NavSTM). The check is in failure model for the refinement of the RHS that specifies the possible behaviours of the state machine navigator ($NavSTM$) except its external events and *tock*. Satisfying this assertion establishes an absence of time locks in the navigator state machine [2].

In using FDR for the verification of the example ANS, as shown in Figure 11 and Figure 12, FDR shows that 109 automatically generated assertions can be verified using the combination of RoboTool and FDR based on the behaviour of the example ANS.

2.7. tock-CSP and Timed CSP

CSP provides a framework for analysis, reasoning and also proof about the behaviour of a system. However, the original version of CSP without the timing information makes it easier to reason about the design and behaviour of systems without considering temporal specifications. But in the case of real-time systems and also timed critical systems, timing is an essential component of the system, which has to be part of the correctness and reasoning about the behaviour of timed systems. Thus, in this case, the original untimed CSP approach is not sufficient for the verification of timed systems. Independent treatment of time is needed to handle timing. This motivates the development of various approaches for extending CSP with timing information, notably *tock-CSP* and Timed CSP.

The time model *tock-CSP* is one of the approaches of extending CSP with timing information for capturing temporal specifications. The approach of *tock-CSP* adds a model of time-frame to measure the progress of time, which is captured with the special event *tock*, forming a version of CSP that models system with their temporal specifications, as discussed in the previous sections.

Another approach of modelling temporal specification in CSP is Timed CSP approach, which adds a sequence of time to describe the timing of each event. The behaviour of a system is modelled with two sequences, a sequence of events and a sequence of real numbers for recording the progress of time. Each element of the timed sequence specifies the time for the corresponding event in the sequence of events. The time in which each event occurs is recorded with a real number. Timed CSP uses the continuous-time model, which is the recommended suitable model of time for modelling real-time systems. In Timed CSP, a trace is a series of events with their corresponding series of time. In the literature [6], timed CSP has been described as the most elegant approach for extending CSP with the temporal constructs.

Unfortunately, this approach complicates the automated verification approach and traces become infinite, which is problematic for verification. At the time of this work, this approach is not supported in the FDR because it is a different model from the original CSP models. Therefore, it needs a new set of theorems and tools, as there currently is no supporting tool for Timed CSP. Additionally, some specifications are difficult to specify with the Timed CSP; for instance, specifying a hard deadline is not possible because there is no way of specifying that an event must happen before a specific time in Timed CSP [6].

Unlike the original CSP, in *tock-CSP* events happen in a given time frame. The time frame is marked with the event tock. Events are recorded sequentially within a time frame, as many as possible finite events occur within a unit time before the next occurrence of the event tock [37]. Thus, a trace contains a sequence of events divided into time frames separated by the event tock. This enables reasoning about the time at which events happen and a time interval between events. This provides a way for expressing urgency, delay and deadline.

The event *tock* is a special event that marks the passage of one unit time, which serves as an interface of transition from one time-frame to another. Also, the event *tock* is a visible event that is neither hidden nor renamed. The behaviour of the event *tock* is not affected by either the system or the environment. All processes synchronise over the event tock. The number of *tock* events in a process quantifies the amount of time progress. The quantification depends on the size of each time frame. There is no minimum separation interval, thus no restriction on granularity.

The approach of *tock-CSP* retains both the original structure and the notations of the original CSP. This makes it easier to both understand and utilise the existing techniques, theories and tools of the original untimed CSP, such as FDR [6, 37]. This includes the constructs of refinement model checking, which check if an implementation process (Impl) model refines the corresponding specification process (Spec) model. Like CSP, *tock-CSP* models can be checked with three options: trace model `([T=)`, failure model `([F=)` and failure/divergence model `([FD=)`. For instance, `Impl` trace-refines `Spec` is express as `Spec [T= Impl`, to mean that the traces of the model Spec contain the traces of the model `Impl` [6,37].

An absence of the event *tock* in a process specifies that the process is completed within one time unit. One shortcoming of the *tock-CSP* approach is that sometimes adding multiple tocks tends to make system descriptions more cumbersome and difficult to understand [37]. The *tock-CSP* models time with a discrete model of a clock, but it has a strong connection with Timed CSP that uses a continuous-time model of a clock [110]. The event *tock* happens regularly with the flow of time. Secondly, with the *tock-CSP* Approach, we are able to differentiate idle (with the process TOCKS), urgent (before the next *tock*) and evolving state (countdown). These three kinds of specifications are quite tricky if not impossible to specify in the untimed CSP [6,37].

In the literature [6,37], it has been recommended that explicit use of the event *tock* to specify the progress of time needs to be sensible and reasonable, because of the tendency to create problems in system description and design, which may lead to building unrealistic systems. For example, this can lead to building a system that may block the progress of time or a system that performs an infinite sequence of events within a finite time. It is recommended and emphasised that a separate check for these abnormalities should be carried out separately [6]. For instance, timing consistency checks using failure divergence is as follows.

```
TOCKS [FD= P\(Events\{tock})
```

As a result of this, *tock-CSP* provides a promising approach for modelling systems with a discrete-time model, which is suitable for automated verification with a well-established CSP tool called FDR; a model checker for automatic analysis and verification of a finite CSP process using refinement checks. FDR uses assertions to check the behaviour of a process. In the case of encountering a failure, FDR provides a counterexample to illustrate the failure. This provides a means of exploring the possible behaviours of a process to check if the process either satisfies or violates its specification. This provides a way of using refinement approach in verifying the specification of systems, including robotics applications.

However, refinement approach with *tock-CSP* has scope and limitations. For instance, *tock-CSP* is not capable of handling liveness specifications. Along this line, it has been prove that, *tock-CSP* using refinement lack the capability and power of expressing liveness specifications. Further investigation shows that, in general capturing temporal logic specifications has to be through refusal testing model. However, due to the difficulty of expressing refusal testing, automatic support becomes problematic. On the other hand temporal logic provides direct constructs for expressing liveness specification. In the next section, we are going to discuss details of TA and temporal logic that we can take advantage in translating *tock-CSP* to TA [7,39,111].

In summary, *tock-CSP* provides a modelling framework that is capable of expressing and analysing timed systems, particularly for event-based systems. It provides a flexible way of specifying synchronisations between a system and its operating environment. Also, it enables reasoning about the behaviour of processes that interact according to the flow of time in discrete format.

2.8. Timed Automata and UPPAAL model checker

A timed automaton is a finite state machine with a finite set of a real-valued clocks. An edge of TA connects a state with another state to form a transition, which may have constraints that include a clock. The constraint enables the transition when it is satisfied with the clock value. A TA reads timed-words that describe the behaviour of a system. In the timed words, each action is associated with a real-valued clock. A set of the accepted timed words for a TA describes the language of the TA, which describes its possible behaviour.

Uppaal model checker is a tool for the modelling and verification of real-time systems. Systems are modelled as a network of the parallel composition of timed automata (TA) that have an external clock. The clock is modelled with a continuous-time model, in which time progresses synchronously with the global clock. Time is quantified with a real number value. Uppaal enables using multiple clocks in a TA. All these clocks progress synchronously together. An individual clock is reset independently, which facilitates modelling temporal specifications, such as delay, timed-budget and urgency [107,112].

In Uppaal models, each process is modelled as a TA that is mathematically defined as a tuple (L, l_0, C, AE, I), where L describes the set of possible locations, l_0 is the set of possible initial locations, C is the set of clocks, A is the set of actions, E is the set of possible edges and I associates invariant with a location. The set of edges $E \subset L \times A \times B(C) \times 2^C \times L$. This describes an edge from location L with an action A, and a guard B(C), with a set of clocks that are reset following the transition to a location L.

The semantics of each TA is defined with a Labelled Transition System (LTS); a tuple $(S, s_0, \rightarrow)$, where $S \subset L \times R^C$, a combination of location and their possible clocks value, s_0 is the initial state, transition $(\rightarrow)$ in the form of $S \times A \times S$. There are two possible transitions. A transition from state to state over either time delay (t) or an action (σ) or combination of both. The operational semantics of TA is defined as follows, given a TA that is defined as (L, l_0, C, A, E, I) with a LTS as $(S, s_0, \rightarrow)$, the two possible transitions are expressed as follows.

1. Delay transition:
   $(l, u) \xrightarrow{d} (l, u + d) \iff (\forall d : 0 \leq d' \leq d \Rightarrow u + d' \in I(l)$
2. Action transition:
   $(l, u) \xrightarrow{a} (l', u') \iff \exists e = (l, a, g, r, l') \in E$
   $such\ that\ u \in g,\ u' = [r \rightarrow 0]u\ and\ also\ u' \in I(l)$

In Uppaal, a TA performs an action over an edge synchronously with the environment and possibly with some other TA. This leads to a transition from one state to another or possibly return to the same state. A state in a system encapsulates all the locations of all the parallel TA, together with their associated clock values and the values of the related variables. In TA, performing an action triggers a transition that changes the state of the whole system. Uppaal uses TA to model a system, with additional features such as clock, template, urgency, committed location, array and user-defined function, as follows.

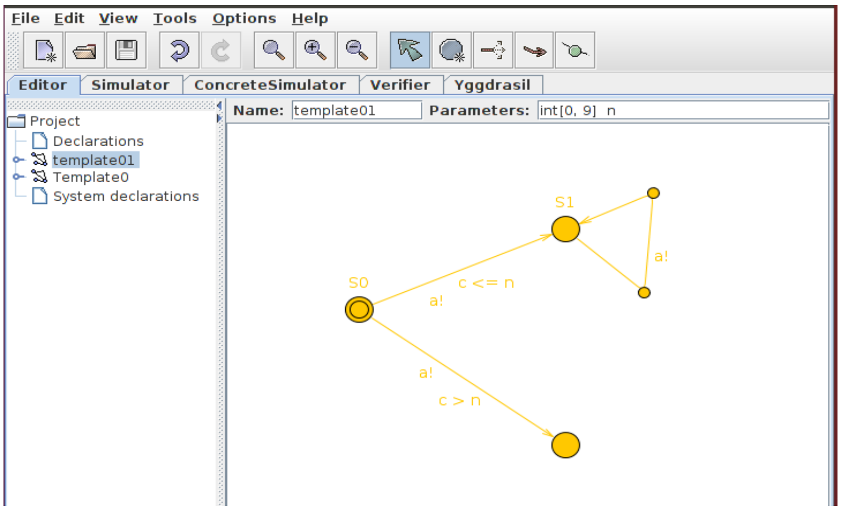
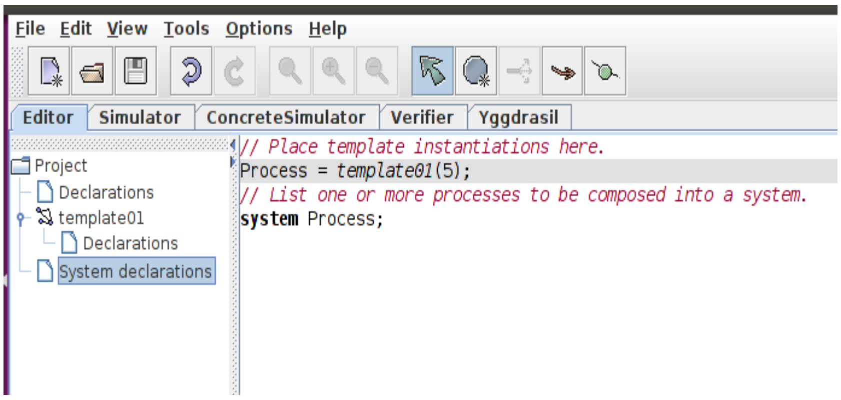

File Edit View Tools Options Help
Editor Simulator ConcreteSimulator Verifier Yggdrasil
Project
Declarations
template01
Template0
System declarations
Name: template01 Parameters: int[0, 9] n
S1
S0
a!
c <= n
a!
a!
c > n

Figure 13: An example of template in UPPAAL


File Edit View Tools Options Help
Editor Simulator ConcreteSimulator Verifier Yggdrasil
Project
Declarations
template01
Declarations
System declarations

```
// Place template instantiations here.
Process = template01(5);
// List one or more processes to be composed into a system.
system Process;
```

Figure 14: Defining a process with a template in UPPAAL

Template of automata: defines a generic automaton that describes a generic process. This concept is similar to the common practice of using a template to define a function in conventional programming languages. A template takes arguments that define the parameters of an automaton.

Example 2.1. An example of using a template in UPPAAL is shown in Figure 13 and also instantiating the template as a process in Figure 14. Figure 13 shows a template of TA that has one argument n, which specifies a deadline for a transition. The deadline is of type integer between 1 and 9. The process performs the event a within the specified deadline; then the process repeats the event a as many times as possible. But, if the event a happens after the deadline, the process terminates with a deadlock.

Variable and Constant Values are the provided constructs for defining values in a *variable* that can be used to define a TA or system. Constant value remains unchanged throughout the execution of a timed automaton. Additionally, a bounded integer variable is provided that defines a variable with a specific range of possible values, which is provided for optimising verification to eliminate checking all other irrelevant possible values and state explosions. For example, a bounded integer variable is declared as $int[min, max]\ varName$. The variable is used like other values to specify a guard, invariants and assignment expressions, as described in the following BNF.

$$
\begin{aligned}
BoundedInteger &::= \text{'}int\text{['} < min > \text{','} < max > \text{']'} \\
max &::= < digit > \\
min &::= < digit > \\
digit &::= 0 \mid 1 \mid 2 \mid 3 \mid 4 \mid 5 \mid 6 \mid 7 \mid 8 \mid 9 \mid < digit >< digit >
\end{aligned}
$$

The terms *min* and *max* represent any positive integer value, including zero. For a valid range, the value *max* has to be greater than the value of *min*. The complete BNF of UPPAAL is provided in Appendix C [7].

Example 2.2. An example of a statement for declaring a variable with a bounded integer in the range 0 to 99 is `int[0, 99]  n;`.

Channel declares an action with the keyword *chan*, and follows with a given identifier of the event. For example, the expression `chan move` declares an action *move* that describes a synchronisation point when an edge fires the action. For example, a TA that performs the action *move*! synchronises with another TA that performs the coaction *move*?; an event that has a similar identifier but labelled with "?". When many possible synchronisation edges are ready, one of them is selected non-deterministically. If none of them are ready, the transition is blocked.

Example 2.3. A statement `chan a;` declares an event *a*. An edge that fires the event *a*! synchronises on this event with a transition label *a*? from another TA.

Broadcast provides a construct for multiple synchronisations. A broadcast channel synchronises more than two TA on a single event. A single TA fires a transition label with a broadcast channel as sender *channel*!. Then all the receiving *channel*? in the other receiving TAs have to synchronise.

Example 2.4. A statement `broadcast chan bc;` declares a multi-synchronisation event *bc*. An edge fires this event with a transition label *bc*!, which synchronises with as many edges as possible on this event using a transition label *bc*?.

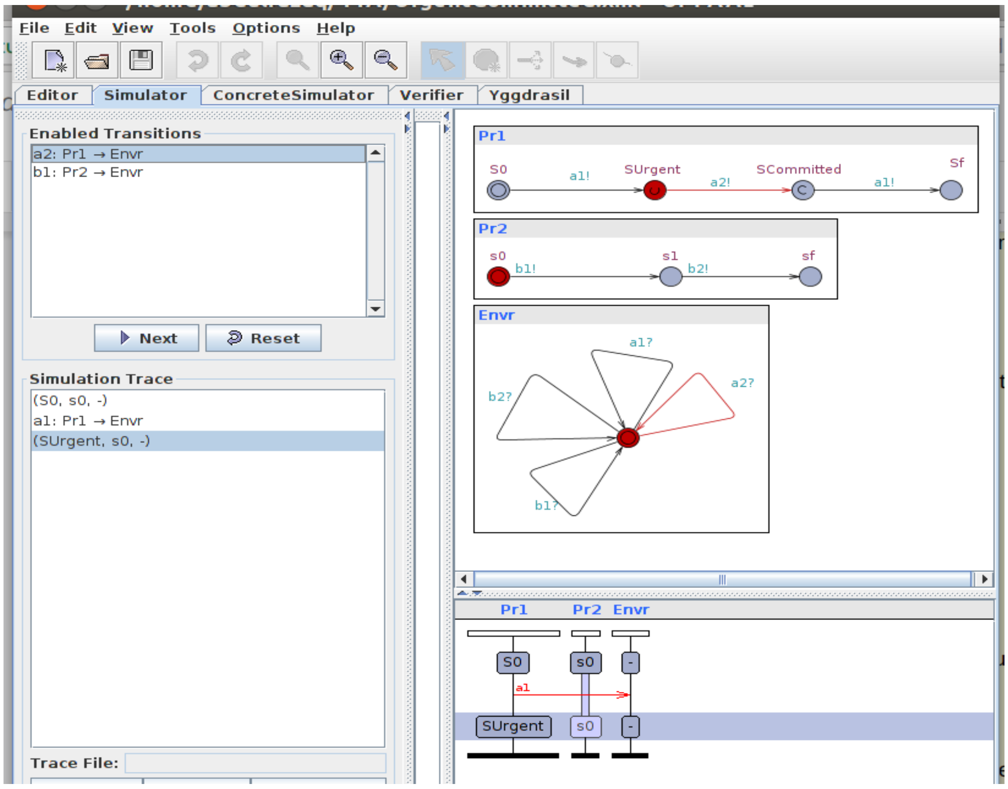
Urgent Channel An urgent channel does not allow any delay on a transition. A transition label with an urgent channel is taken as soon as the transition becomes available. Any transition label with the urgent channel cannot have any guard. Also, the urgent

[7]The BNF is part of the documentation for a parser of the UPPAAL tool

channel is declared using two keywords *urgent* and *chan*. For instance, the expression `urgent chan brake` declares an urgent channel name brake. A transition labelled with the event *brake* is an example of an urgent transition.

Urgent Location An urgent location is a location that does not allow any delay. This location is similar to a location that has an implicit clock invariant of $x \leq 0$. This means that the process leaves the location instantly. The UPPAAL IDE provides options for declaring a location as an initial, urgent or committed location.

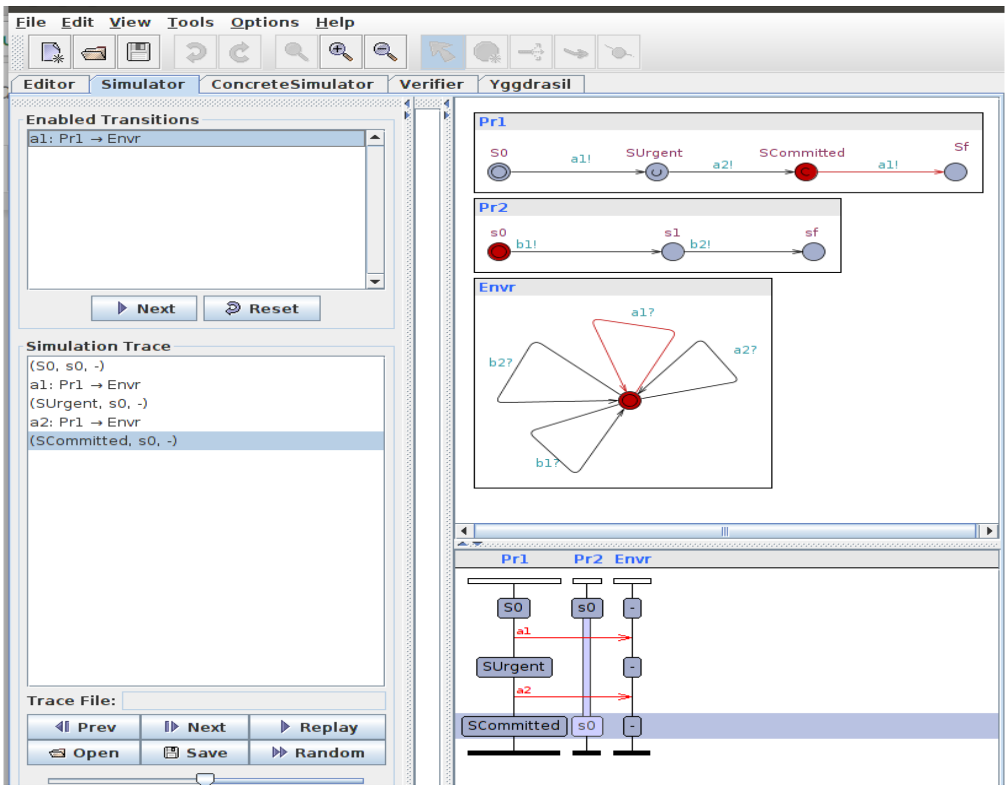
Committed Location is superior to urgent location because, in addition to being an urgent location, it also blocks any other transition except its transition or any transition from another committed location. When a process reaches a committed location, it only progresses from one of the outgoing edges of that location. Example 2.5 illustrates the differences between urgent and committed locations, as follows.



```
Enabled Transitions
a2: Pr1 → Envr
b1: Pr2 → Envr

Simulation Trace
(S0, s0, -)
a1: Pr1 → Envr
(SUrgent, s0, -)

Trace File:
```

Figure 15: A system in an urgent location

Example 2.5. In illustrating the differences between the two types of locations, urgent and committed, we use three processes: *Pr*1, *Pr*2 and *Envr* to model a system. A TA that models the process is shown in Figure 15. The process *Pr*1 performs the event *a*1, *a*2, *a*1 and then deadlocks at the final state *sf*. The second process *Pr*2 performs



Enabled Transitions

```
a1: Pr1 → Envr
```

Simulation Trace

```
(S0, s0, -)
a1: Pr1 → Envr
(SUrgent, s0, -)
a2: Pr1 → Envr
(SCommitted, s0, -)
```

Figure 16: A system in a committed location

the action $b1$, $b2$ and *deadlocks* at the final state sf. The third process *Envr* models the environment that enables all the events $a1$, $a2$, $b1$ and $b2$.

In Figure 15, when the system is in the urgent location, the possible transitions are shown in the top left window of Figure 15. In this case, the two possible transitions are $a2$ and $b1$; $a2$ on the outgoing edge of the urgent location of the TA $Pr1$, and $b1$ on the outgoing edge of location $s0$ of the second TA $Pr2$. But when the system reaches the committed location, as shown in Figure 16, the only available transition is $a2$, which is the only outgoing edge in the committed location of the TA $Pr2$.

Array construct is provided for declaring multiple clocks, channels, constants and integer variables. The index value specifies the size of the array, similar to the well-known array in the data structure.

Example 2.6. An expression of the form *chan ch*[5]; declares five channels, namely ch[0], ch[1], ch[2], ch[3] and ch[4]. The appended number describes the index of the variables.

There are other constructs such as the structure *record* for storing multipart data, which is similar to the syntax of the C programming language. A record is declared with the keyword *struct*. Additionally, custom data type and user-defined function are also available with the same structure of C programming language, except that the pointer is not allowed in UPPAAL.

Guard Expression Guard is a special expression that evaluates to a Boolean value. Only constants, integer variables and clocks are allowed in expressing a guard. Essentially a guard consists of clocks or clocks compared to an expression. An expression guard is restricted to using side-effect-free function, which returns a Boolean value. Also, UPPAAL enables the use of multiple conjunctions of guards to specify a compound guard.

Update Expression assigns a new integer value to a clock or variable. A single transition can update multiple expressions, which include clock or variables.

Invariant is a side-effect-free expression that evaluates to a Boolean value. In defining an invariant, the only acceptable constructs are clock, integer constant, integer variables and side-effect-free function.

Uppaal query UPPAAL models a system with TA and a query expression is used to check specifications of interest in the system. The query is expressed using a subset of the simplified version of TCTL. The query has two parts: the state formula and path formula. The path formula quantifies the interesting path or trace of a specification, which is expressed using the four symbols: $A, E, []$ and $<>$, where: A specifies all paths, E specifies some paths, [] specifies all states in the path and $<>$ specifies some states in the path. A state formula is a side-effect-free expression that evaluates to a Boolean value. A special keyword *deadlock* is provided for checking deadlock freedom.

The BNF for the formula for expressing query is as follows:

$$\begin{aligned}Querry ::= {} & A[] < property > | A <>< property > | E[] < property > | \\ & E <>< property > | < property > \rightsquigarrow < property >\end{aligned}$$

$$\begin{aligned}property ::= {} & \text{automata.location} | \text{data-guard} | \text{clock-guard} | \\ & < property > \; and < property > | < property > \; or < property > | \\ & not < property > | < property > \; imply < property > | (< property >)\end{aligned}$$

The term *automata.location* specifies a location of an automaton. The terms $data - guard$ and $clock - quard$ are guard expressions involving variable and clock, respectively. In specifying a query for verification, UPPAAL supports three classes of path formulæ, reachability, safety and liveness.

Reachability checks the existence of a path from the initial state to a particular state that satisfies a given condition.

Example 2.7. $E <> car.exit$
The process *car* would reach a location *car.exit*.

Safety checks a condition that is invariantly true in all reachable states. This is specified as A[] *condition* or E [] *condition*.

Example 2.8. `A[]` φ
The condition φ is true in all reachable states.

Example 2.9. `A[]` $(count < 3)$
This specifies that a property $count < 3$ is true in all states along any path.

Example 2.10. `E[]` φ
Existence of complete path with the condition φ always true

Example 2.11. `E[]` $(ck < 3)$
There exists a complete path in which the clock $ck < 3$ is true in all the states along the path.

Liveness checks a condition that is eventually satisfied. This is expressed as `A<> condition`. Another alternative form for expressing liveness property is *'lead to'*, which checks a state that eventually leads to another state. For instance, *A leads to B* is expressed as *A --> B*. This means that when the system is in the first state *A* happens then eventually the system will be in the second state *B*.

Example 2.12. `A<>`φ
The condition φ will eventually become true along any path.

Example 2.13. `A<>` $(count < 3)$
The property $count < 3$ is true in at least one state along any path.

Example 2.14. *car.enter --> car.exit* :
This statement specifies that whenever a TA of the process *car* reaches a state *car.enter*, it will eventually reach a state *car.exit*.

2.9. Final Considerations

In summary, in this section, we present literature for the software development process for robotic systems, which has been discussed to address the need for tools and techniques to support the development of robotics systems. DSL and MDE have been recognised as the two major SE principles that are indispensable to robotics. The discussion of DSMLs describes both the concept of DSML and interesting desirable features that are useful for developing robotic systems.

Sample DSMLs have been discussed, identifying the various interesting features that they provide. Overall, we discussed that there are many features for enhancing the developmental process of the robotics application, but it is clear that the verification aspect receives less attention. This indicates the need for equipping the verification process with rigorous formal `V&V` techniques that are suitable for robotic applications.

Considering the studied DSMLs, it is easy to see that most of them support and adopt graphical notations and the idea of generating executable logic from the models. From this study, we found that state machine notation is the most commonly used for modelling behaviour. The framework EMF is the favoured framework for graphical modelling. Various approaches to formalisation and architectural style were demonstrated. There is a common use of discrete approximation for handling continuous-time models. In the case of handling uncertainties, only one DSML RoboChart provides the means for modelling probability. Lastly, none of these DSMLs are proprietary and also none of them provide techniques for generating test cases.

For the industrial use of these DSMLs, no information was found be available available on how industries are using these DSMLs and the experience they have in using them. Recently, RobotML [12] was extended with features to increase its fitness for industrial use, but at this time, useful information is yet to be available for industrial use of RobotML. This is an interesting aspect to explore in future research.

Limitations have been highlighted in Section 2.4 that made both GenoM and DSML for adaptive systems unsuitable for modelling continuous-time, particularly towards improving formal techniques. On the other hand, RoboChart stands apart as an evolving graphical language with a good foundation for the comprehensive use of the formal method since its inception. As indicated in Tables 1–4, RoboChart has a good plan for using formal method as well as interesting features suitable for both robotic system development and verification. This provides a suitable foundation for improving the language in the aspect of temporal reasoning for robotic systems. This section includes a discussion of the provided constructs of RoboChart for modelling a system, which provides a way of modelling and verifying temporal specification.

Automated tools are the main vehicles that make formal techniques practical for use in verifying real systems, particularly in the verification of large system like robotics applications. As discussed in this section, there are two popular modelling approach refinement and temporal logic. Both of these supporting tools have strengths and weaknesses for verifying software applications. Based on the literature review, we find that the facilities of UPPAAL can complement the strength of FDR in verifying temporal specifications, particularly verifying specifications that can not be verify with refinement modelling approach [39].

In this work, we analyse using the facilities of UPPAAL to verify temporal specifications of *tock-CSP*, which will also provide better techniques for supporting the existing works that are based on *tock-CSP*, such as RoboChart for verifying robotics applications. The next section describes the technique we developed for using the UPPAAL tool to verify *tock-CSP* model, which will enhance the verification of robotics applications.

Chapter Three

3. Translation Technique

This chapter describes the technical part of this research. We discuss the technique we developed for translating *tock-CSP* models into Uppaal models. First, we begin with characterising *tock-CSP* as a language for capturing the semantics of RoboChart models. Second, we describe our strategy for developing the translation technique. Third, we present the translation rules for translating *tock-CSP* into Timed Automata (TA). Finally, we discuss a justification of the translation rules using trace semantics, and then conclude the chapter with a final consideration of the translation technique.

In Section 3.1, we characterise *tock-CSP* as a language for modelling temporal specifications. This section presents a BNF for the language, together with well-formedness conditions for the BNF. As part of this work, we implement an Abstract Syntax Tree (AST) of the BNF using Haskell, which we use for both descriptions of the translation rules and also an implementation of the translation technique. Listing 3.1 presents the relevant part of the Haskell implementation of the AST.

In this work, we consider translating *tock-CSP*, which captures the semantics of the RoboChart. This is because RoboChart is at the development stage during this work and it will continue to evolve that may lead to changing its structure by the time we complete this work. However, the semantics of the structure will always confirm to the constructs of *tock-CSP* as used in this translation work. As such, our work will apply to the RoboChart despite its continues development. Besides, this decision expands the application of our work to cover other relevant work around *tock-CSP*, such that any work based *tock-CSP* will benefit from our work.

In Section 3.2, we describe the strategy we follow in developing the translation technique. We describe our approach of using small sizes TA to capture the semantics of *tock-CSP*. To ease understanding of the approach, we provide an example that demonstrates the translation strategy in translating *tock-CSP* processes into Uppaal models.

In Section 3.3, we describe the translation rules we develop for translating *tock-CSP* models into suitable Timed Automata for Uppaal. For each construct of the BNF, we provide a rule for translating the construct into TA. For concise presentations of the rules, we used Haskell code in precisely presenting the translation rules. Also, we provide examples that illustrate using each rule in translating *tock-CSP* processes. Additional examples are also provided in Appendix D.

3.1. Characterisation of tock-CSP

This section describes the characterisation of *tock-CSP* using BNF grammar. The following BNF in Figure 17 defines a valid syntax for constructing a *tock-CSP* process that we consider within the scope of this work, then, follow with Haskell implementation of the BNF in Definition 3.1.

$$
\begin{aligned}
NamedProc ::=\ & Name\ CSPproc \\
\mid\ & Name\ CSPexpression\ CSPproc
\end{aligned}
$$

$$
\begin{aligned}
CSPproc ::=\ & STOP \\
\mid\ & Stopu \\
\mid\ & SKIP \\
\mid\ & Skipu \\
\mid\ & Wait(Expression) \\
\mid\ & Waitu(Expression) \\
\mid\ & Event \rightarrow CSPproc \\
\mid\ & CSPproc \mathbin{\Box} CSPproc \\
\mid\ & CSPproc \sqcap CSPproc \\
\mid\ & CSPproc\ ;\ CSPproc \\
\mid\ & CSPproc \mathbin{|||} CSPproc \\
\mid\ & CSPproc \underset{\{Event\}}{\parallel} CSPproc \\
\mid\ & CSPproc \triangle CSPproc \\
\mid\ & CSPproc \overset{d}{\triangleright} CSPproc \\
\mid\ & CSPproc\ \Theta_{\{Event\}}\ CSPproc \\
\mid\ & CSPproc \setminus \{Event\} \\
\mid\ & CSPproc[\{Event\}/\{Event\}] \\
\mid\ & EDeadline(Event, Expression)
\end{aligned}
$$

$$
Event ::= eventIdentifier \mid tock
$$

Figure 17: BNF of *tock-CSP* for the translation technique

The BNF is implemented into AST using Haskell in the following Definition 3.1.

Definition 3.1. Data definition of `CSPproc`

```
data CSPproc = STOP
             | Stopu
             | SKIP
             | Skipu
             | WAIT         Int
             | Waitu        Int
             | Prefix       Event    CSPproc
             | IntChoice    CSPproc  CSPproc
             | ExtChoice    CSPproc  CSPproc
             | Seq          CSPproc  CSPproc
             | Interleave   CSPproc  CSPproc
             | GenPar       CSPproc  CSPproc [Event]
             | Timeout      CSPproc  CSPproc Int
             | Interrupt    CSPproc  CSPproc
             | Exception    CSPproc  CSPproc [Event]
             | Hiding       CSPproc  [Event]
             | Rename       CSPproc  [(Event, Event)]
             | Proc         NamedProc
             | EDeadline    Event    Int
             | ProcID       String
```

In the following explanation, we use two metavariables P and Q, and decorations on these names, to denote elements of the syntactic category *CSPproc*. We use the symbol e to represent an element of the set *Event*. Also, the symbols A and B are used to represent set of events. Lastly, the parameter d represents a *CSP* expression that evaluates to a positive integer [8].

STOP specifies a process at a stable state in which only the event tock is allowed to happen. This means that the process *STOP* enables passage of time only, no other events are allowed to happen.

Stopu specifies a process that immediately deadlocks. Unlike the previous process *STOP*, this process *Stopu* does not allow any time to pass before the deadlock.

SKIP specifies a process that reaches a successful termination point, where it can either terminate or allow time to pass using the event *tock* before termination. In

[8]Additional details is available in `https://www.cs.ox.ac.uk/projects/fdr/manual/cspm/syntax.html#csp-expressions`

essence, only two events are possible at that state, *tock* for time or *tick* for termination.

$Skipu$ specifies a process that immediately terminates. Unlike the previous process $Skipu$, this process does not allow time to pass before termination. In essence, the process immediately performs the termination event *tick*.

$WAIT(d)$ specifies a delay process that remains idle for a certain amount of unit time d. After the idle time elapses, either the process terminates with the event *tick* or allows arbitrary units of times to pass before termination.

$Waitu(d)$ specifies an urgent delay process that remains idle for a fixed amount of unit time d. The process terminates immediately after the fixed delay time d.

$e \rightarrow P$ Prefix describes a process that offers to engage with an event e and then subsequently perform the behaviour of the process P.

$P \sqcap Q$ Internal choice specifies a process that has different autonomous choices of behaviour, P and Q. Independently the process $P \sqcap Q$ behaves either as P or Q, regardless of the choice of the environment. In the case of this internal choice, the environment has no control over the two possible choices of P and Q.

$P \Box Q$ External choice specifies a process that is ready to engage in the behaviour of either P or Q depending on the choice of the environment. The process offers to engage with the initials of both P and Q, for each chosen initials the process $P \Box Q$ provides the corresponding behaviour of either process P or Q. In the case of this external choice, the environment has control in choosing the behaviour of the process. This is the complement of the previous internal choice where the process has control over the choice of the behaviour.

Well-formedness In the case of external choice, there is a restriction that the event *tock* is not allowed to appear in the initials of either of the processes in external choice. That is $tock \notin (initials(P) \cup initials(Q))$. This is because having the event *tock* as part of the initials will cause non-determinism between the process behaviour and progress of time.

$P; Q$ Sequential Composition specifies a composition of two processes P and Q that run one process after the other. The first process P begins until it terminates, then follows with the behaviour of the subsequent process Q.

$P|||Q$ Interleaving specifies a parallel composition where both the processes run independently without any interaction. In this case, the processes have no common interaction points except for the termination point. Interleaving processes do not synchronise in any of their events.

Well-formedness Implicitly, the processes P and Q have to match the flow of time. If both processes perform the time event *tock*, they synchronise with the flow of time on the event *tock*, which implies that the two processes implicitly synchronise on the flow of time and the time event *tock*.

$P \underset{A}{\|} Q$ Generalised parallel specifies a parallel composition of two processes P and Q that run in parallel and synchronise on a specified set of events A. Independently, each of the processes performs its events that are outside the set A.

Well-formedness The set A implicitly contains the event *tock*.

$P \overset{d}{\triangleright} Q$ Timeout delay specifies a composition of two processes P and Q, where a deadline d is specified for the first process P to engage with performing an event from its initials $initials(P)$. If the first process P engages, then the whole process behaves as the process P. After the deadline d time unit, if the first process P did not engage, the second process Q takes over the control, and the whole process behaves like the second process Q.

Well-formedness The expression d is restricted to an expression that evaluates to a natural number. This is because *tock-CSP* is based on a discrete-time model that records time-progress with discrete-event tock. Also, the processes P and Q are not allowed to begin with the timed event *tock*.

$P \triangle Q$ Interrupt operator describes a process P that can be interrupted by another process Q at any time during the execution of P. The first process P runs until the second process Q performs a visible event. Whenever the second process performs an external action, it interrupts the execution of the first process. The interrupted process is blocked, and the second process takes over the control, then the whole process behaves like the second process Q. If the second process Q did not perform an event, the entire process behaves as the first process P up to its completion.

Well-formedness There is a restriction that the event *tock* is not allowed to be in the initials of the second (interrupt) process. This means that an interruption cannot begin with the event tock, because the time event *tock* causes non-determinism between the interrupt and the process of time.

$P\ \Theta_{\{Event\}} Q$ The operator exception describes a process that begins until the right-hand-side process performs an exceptional event to start the left-hand-side process. So, the process P begins until it performs an event from the exceptional events *Event* that triggers the process Q.

$P \setminus A$ Hiding specifies the behaviour of a process P which hides all the events in set A. The hidden events A becomes a special event *tau* that are not visible to the environment, as such the environment has no control over the hidden events.

Well-formedness In the case of hidden, there is a restriction that hidden events should not include the time event *tock*. This is because a process should not control the progress of time.

$P\ [A/B]$ Renaming specifies a process that renames a list of its events A with corresponding names of events in list B, in one to one mapping. The renaming operator transforms a process into another process with the same structure but appears with different names of the renamed events A.

Well-formedness Here, the restriction is that the event *tock* should not be renamed to another event, and no other event can be renamed to be *tock*. This is because the time-event tock is a special event dedicated to recording the progress of time.

$Edeadline(e, d)$, the process event deadline specifies a process that must perform the event e within the deadline d. So, the event e must happen within the deadline d.

3.2. Strategy for the Translation Technique

The translation technique produces a list of small TAs [9], such that the occurrence of each *tock-CSP* event is captured in a small TA with an UPPALL action, which records an occurrence of an event. The action has the same name as the name of the translated event from the input *tock-CSP* process. Then, the technique composes these small TA into a network of TA that express the behaviour of the original *tock-CSP* model. The main reason for using small TA is coping with the compositional structure of *tock-CSP*, which is not available in TA [113]. The small TA provides enough flexibility for composing TA in various ways that capture the behaviour of the original *tock-CSP* process.

The connections between the small TAs are developed using additional coordinating actions, which coordinate and link the small TA into a network of TA to establish the flow of the translated *tock-CSP* process. Each coordinating action `a!` (with an exclamation mark) synchronises with the corresponding co-action `a?` (with a question

[9] In this work, we use TA that has few states and transitions connected together into a network of Uppaal models

mark) to link two TAs, in such a way that the first TA (with `a!`) communicates with the second TA that has the corresponding co-action (`a?`), in the form of synchronous communication.

Definition 3.2. Coordinating Action

A coordinating action is an UPPAAL action that is not part of the original *tock-CSP* process. There are six types of coordinating actions:

- **Flow action** only coordinates a link between two TAs for capturing the flow of the behaviour of the original *tock-CSP* process.
- **Terminating action** records termination information, in addition to coordinating a link between two TA.
- **Synchronisation action** coordinates a link between a TA that participates in a multi-synchronisation action and a TA for controlling the multi-synchronisation.
- **External choice action** coordinates the translation of external choice such that choosing one of the processes composed with external choice blocks the other alternative choices.
- **Interrupt action** initiates an interruption of a process that enables a process to interrupt other processes that are composed with an interrupt operator.
- **Exception action** coordinates a link between a TA that raises an exception and a control TA for handling the exception.

The name of each coordinated action is unique to establish correct flow. The name of a flow action is in the form `startIDx`, where `x` is either a natural number or the name of the original *tock-CSP* process. Likewise, the name of the remaining coordinating action follows in the same pattern `keywordIDx` where `keyword` is a designated word for each of the coordinating actions: `finish` for terminating action, `ext` for an external choice action, `intrp` for an interrupting action, and `excp` for an exception action. Similarly, the name of a synchronising action is in the form `eventName___sync`, that is an event name appended with the keyword `___sync` to differentiate the synchronisation event from other events.

Termination actions are provided to capture essential termination information from the input *tock-CSP* in the cases where a TA needs to communicate a successful termination for another TA to proceed. For example, as in the case of sequential composition `P1;P2` where the process `P2` begins after successful termination of the process `P1`.

For each translated *tock-CSP* specification, we provide an environment TA that has corresponding co-actions for all the translated events of the input *tock-CSP* process. In addition, the environment TA has two coordinating actions that link the environment TA with the network of the translated TA. First, a flow action that links the environment with the first TA in the list of the translated TA. Also, this first flow action is the starting action that activates the behaviour of the translated TA. Second, a terminating action that links back the final TA in the list of the translated TA to the environment TA, and also records a successful termination of the whole process.

Definition 3.3. Environment TA

An environment TA models an explicit environment for Uppaal models. The environment TA has one state and transitions for each co-action of all the events in the original *tock-CSP* process, in addition to two transitions for the first starting flow action and the final termination co-action.

Example 3.1. A simple example that illustrates a translation of an Automatic Door System (ADS1), which opens a door, and then after at least one-time unit [10], the system closes the door. A *tock-CSP* process for modelling a simple version of ADS is:

```
ADS1 = open -> tock -> close -> SKIP
```

Translation of the process ADS1 produces the following list of TA in Figures 18 – 22.

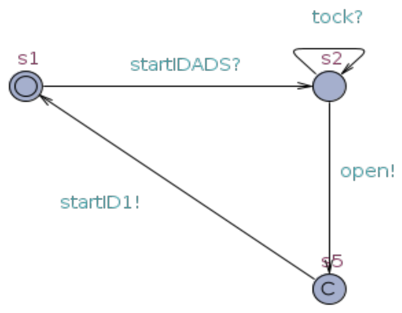


Figure 18: TA1 for the translation of the event `open` in the process ADS1.

[10]The models of *tock-CSP* never blocks time, as such each event `tock` models at least one or more unit time. In translating the event `tock`, we use a recursive transition to capture the progress of time for at least one time unit.

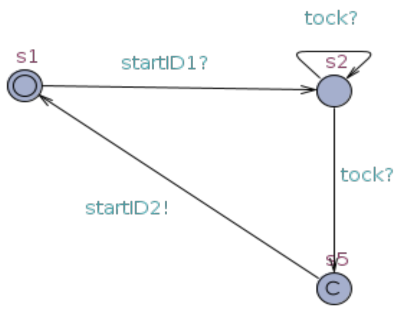


Figure 19: TA2 for the translation of the event `tock` in the process ADS1.

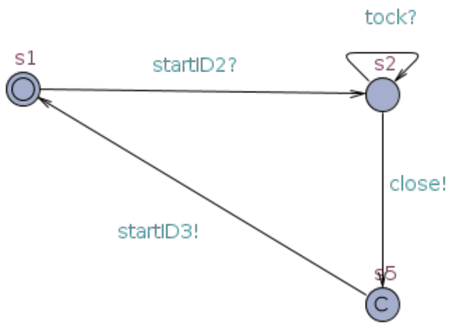


Figure 20: TA3 for the translation of the event `close` in the process ADS1.

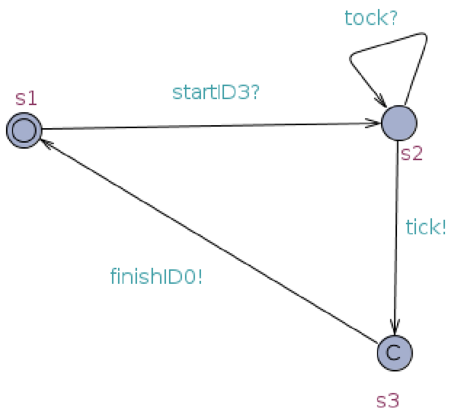


Figure 21: TA4 for the translation of the termination event `tick` in the process ADS1.

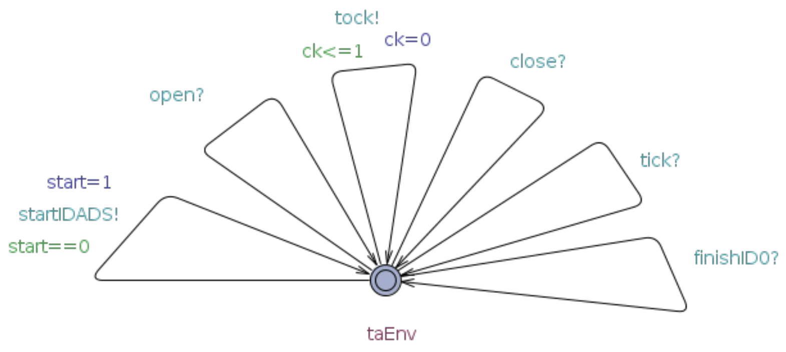


Figure 22: TAenv is an explicit environment of the UPPAAL model ADS1

In Example 3.1, we illustrate a translation of *tock-CSP* into a network of TA in Figure 18 – 22. The *tock-CSP* process ADS01 models the behaviour of an automatic door that opens a door and then at least after one unit time the system closes the door. Later on, we will extend the example to include synchronisation.

Translating Multi-synchronisation

In translating multi-synchronisation events, we adopt a centralised approach developed in [114] and implemented using Java in [115]. The approach describes using a separate centralised controller for controlling multi-synchronisation events. Here, we use UPPAAL broadcast channel to communicate synchronisation between the synchronisation TA and the TAs that participate in the synchronisation. In the following Example 3.2 we illustrate the strategy of translating synchronisation. Then, Definition 3.4 provides a definition of the synchronisation TA.

Definition 3.4. Synchronisation TA

A synchronisation TA coordinates synchronisation actions. Each synchronisation TA has an initial state and a committed state for each synchronisation action, such that each committed state is connected to the initial state with two transitions. The first transition from the initial state has a guard and an action. The guard is enabled when all the processes are ready for the synchronisation, which also enables the TA to perform the associated action that notifies the environment of its occurrence. In the second transition, the TA broadcasts the synchronisation action to all the processes that synchronise on the synchronisation action.

When all the participating TA become ready, a synchronisation TA broadcasts the multi-synchronisation action such that all the corresponding participating TAs synchronise using their corresponding co-action. The provided guard ensures the TA synchronises with the required number of TAs that participate in a multi-synchronisation action. The guard blocks the broadcast multi-synchronisation action until all the participating TAs become ready, which enables the corresponding guard for broadcasting the multi-synchronisation action.

Example 3.2. An extended version of the Automatic Door System (ADS2) opens a door, and after at least one time unit, closes the door in synchronisation with a lighting controller, which turns off the light. In *tock-CSP*, its description is as follows.

```
       ADS2 = Controller [|{close}|] Lighting

Controller = open -> tock -> close -> Controller

  Lighting = close -> offLight -> Lighting
```

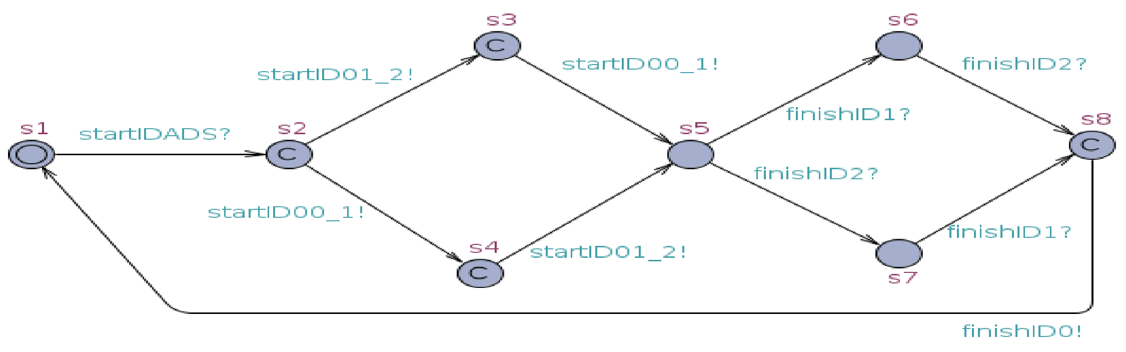
For example, a translation of the process ADS, from Example 3.2, produces the network of small TAs in Figure 23. Details of the translated TA are as follows. Starting from the top-left corner, the first TA captures concurrency by starting the two concurrent automata corresponding to the processes `Controller` and `Lighting` in two possible orders, either `Controller` then `Lighting` or vice versa, depending on the choice of the operating environment. Afterwards, it also waits on state `s5` for their termination actions in the two possible orders, either `Controller` then `Lighting` or vice versa, depending on the process that terminates first. Then, after the termination of the second process the whole system terminates with the action `finishID0`, if both process terminate.

The second (TA02), third (TA03) and fourth (TA04) TAs capture the translation of the process `Controller`. TA02 captures the occurrence of the event `open`. TA03 captures the occurrence of the event `tock?` (with a question mark) for synchronising with the environment in recording the progress of time. TA04 captures the occurrence of the event `close`, which synchronises with the synchronising controller, the fifth TA (TA05).

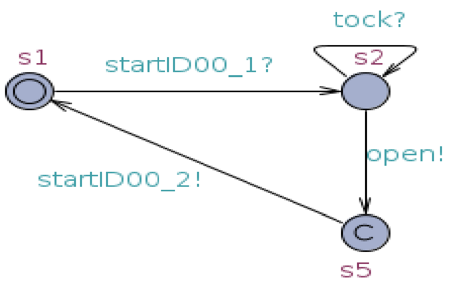
The sixth (TA06) and seventh (TA07) capture the translation of the process `Lighting`. TA06 captures the translation of `close`, which also needs to synchronise with the synchronisation controller (TA05). TA07 captures the event `offLight`. Finally, the last TA (TA08) is an environment TA that has co-actions for all the translated events. Also, the environment TA serves the purpose of 'closing' the overall system as required for the model checker. In the environment TA, we use the variable *start* to construct a guard *start* == 0 that blocks the environment from restarting the system. This concludes the description of the list of TA for ADS.

Translating Interrupt

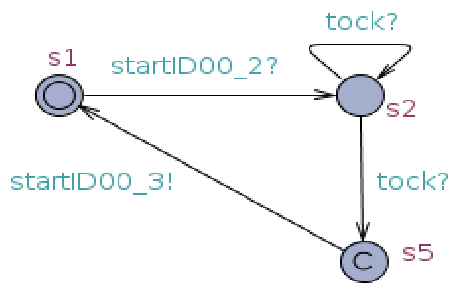
In *tock-CSP*, a process can be interrupted by another process when the two processes are composed with an interrupt operator (/\). This is due to the compositional structure of *tock-CSP*. However, in the case of TA, an explicit transition is needed for expressing an interrupt, which enables a TA to interrupt another one. So in this translation work, we provide an additional transition for capturing an interrupt event using an interrupt action, as defined in the coordination actions (Definition 3.2).



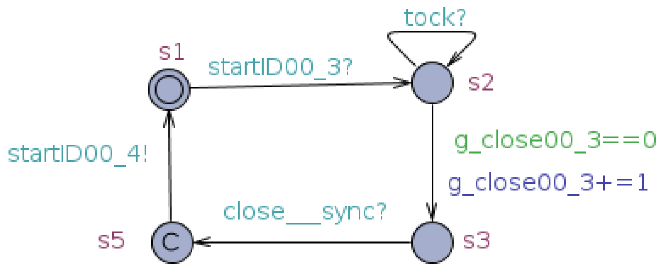
(a) TA01



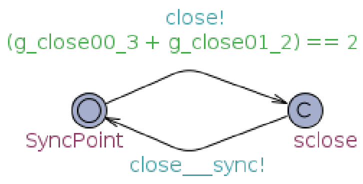
(b) TA02



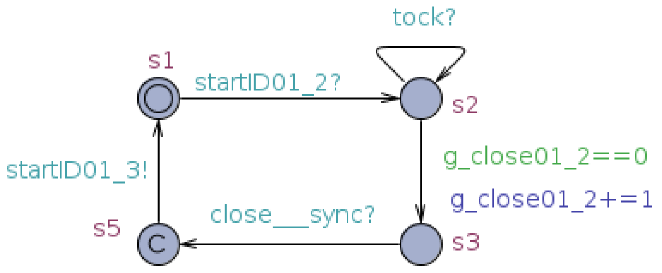
(c) TA03



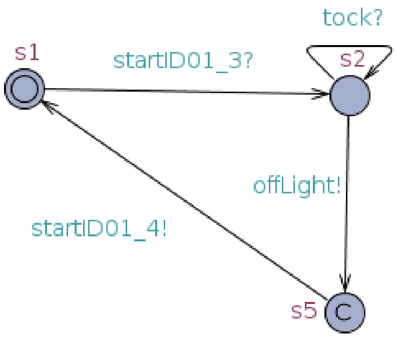
(d) TA04



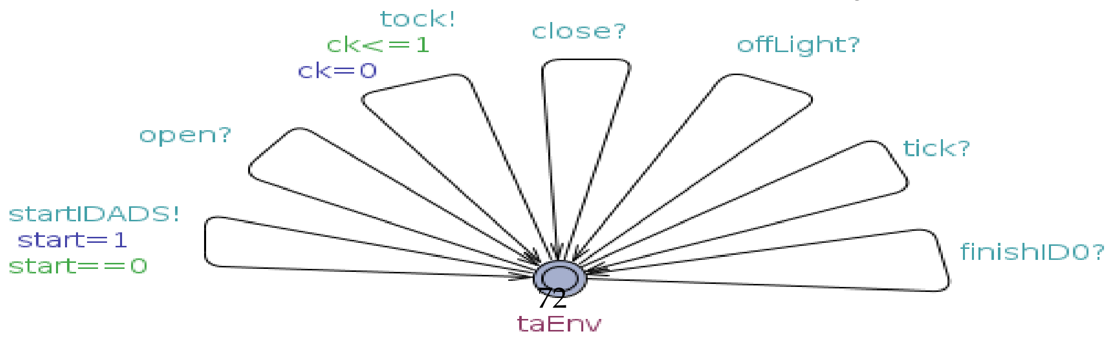
(e) TA05



(f) TA06



(g) TA07



(h) TA08

Figure 23: A list of networked TA for the translation of the process ADS2.

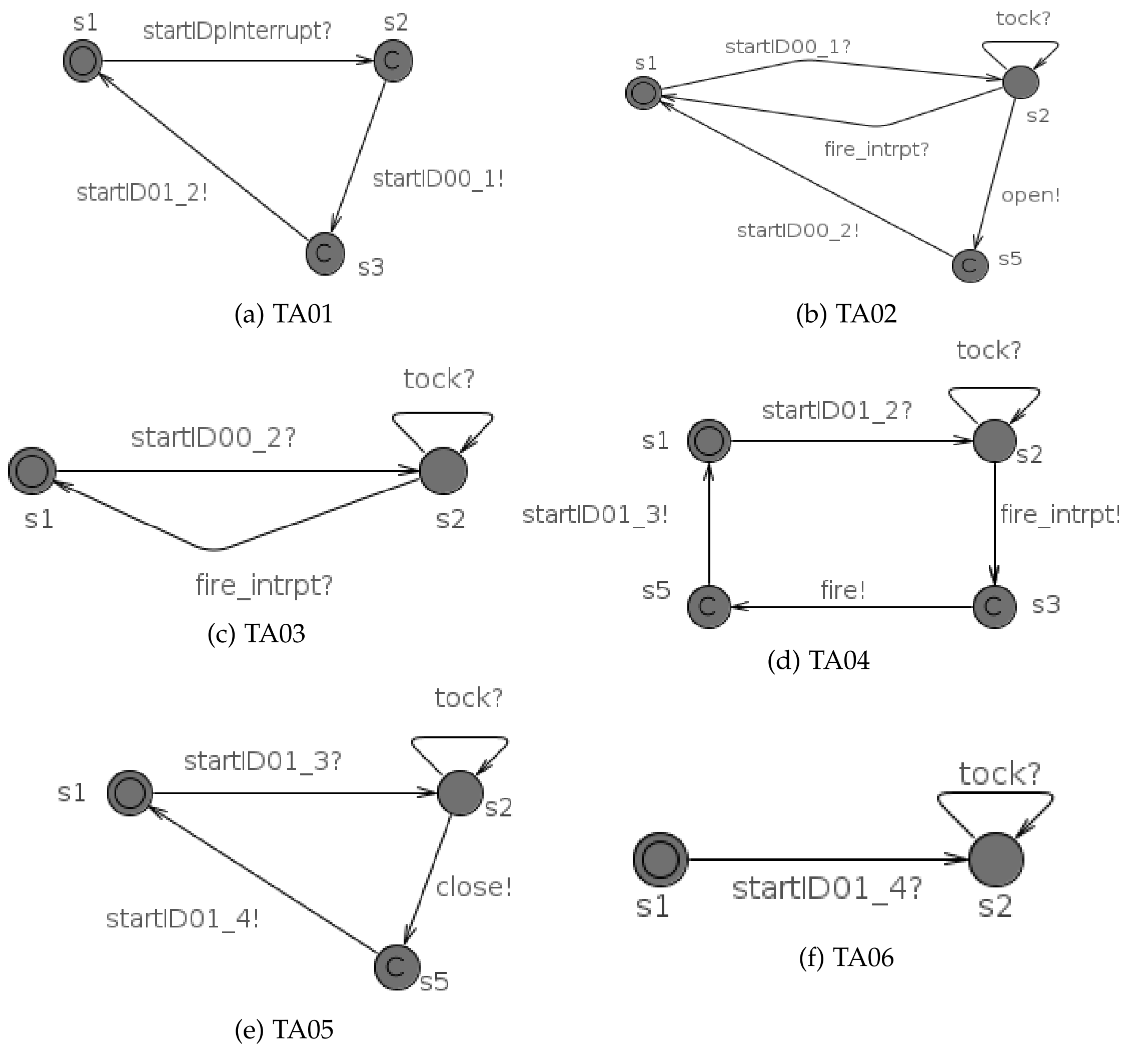


Figure 24: A list of TA for the translated behaviour of the process `Pi`.

For instance, given a process `Pi = P1/\P2`, the process `P1` can be interrupted by process `P2`. Thus, in translating each event of process `P1`, we provide additional transitions for the initials of the interrupting process `P2`, which enables the translated behaviour of `P2` to interrupt the translated behaviour of `P1` at any event.

An example of translating interrupt is provided in Figure 24, which illustrates a translation of the process `Pi = (open->STOP)/\(fire->close->STOP)`. For the process `Pi`, the RHS process `(fire -> close -> STOP)` can interrupt the behaviour of `(open->SKIP)` at any stable state. In the translated behaviour of the LHS process, we provide additional interrupting actions (`fire_intrpt`) that enable the translated behaviour of the RHS process to interrupt the LHS process. The interrupting actions are provided only for the initials of the RHS process (`fire`).

In Figure 24, starting from the top-left, TA01 is a translation of the operator interrupt. The second and third, TA03 and TA04 capture the translation of the LHS process `open->STOP`. TA04, TA05 and TA06 are translation of the RHS process `fire->close->STOP`. The environment TA is also omitted because it is similar to the last TA in Figure 23.

The first TA starts both processes using `startID00_1!` and `startID01_2!`, respectively. The second TA synchronises on the flow action `startID00_1` and moves to location `s2` where the TA has 3 possible transitions for the actions: `tock?`, `open!` and `fire_intrpt?`. With the co-action `tock?`, the TA records the progress of time and remains on the same location `s2`. With the co-action `fire_intrpt?`, the TA is interrupted by the RHS, and it returns to its initial location `s1`. With the action `open!`, the TA progresses to location `s5` to perform the flow action `startID00_2`, which activates the third TA for the subsequent process `STOP`.

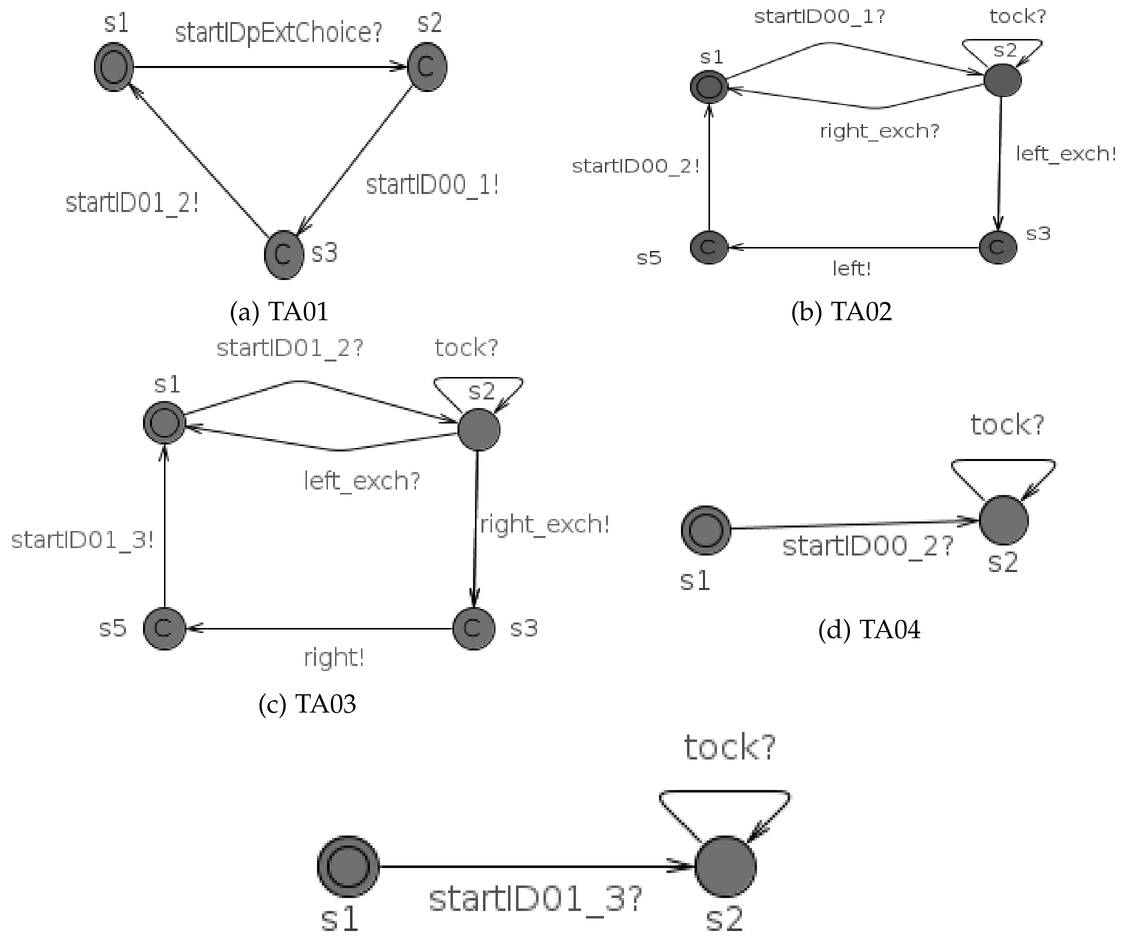
The third TA03 synchronises on the flow action `startID00_2` and moves to location `s2`, where it either performs the action `tock?` to record the progress of time or is interrupted with the co-action `fire_intrpt?`, and returns to its initial location `s1`. This completes the behaviour of the process `open->STOP`.

The fourth TA04 is a translation of the event `fire`. The TA begins with synchronising on the flow action `startID01_2`, which progresses by interrupting the LHS process using the interrupting flow action `fire_intrpt`, then `fire`, and proceeds to `startID01_3` for starting TA05. which synchronises on the flow action and moves to location `s2`, where it either performs the action `tock?` for the progress of time and remains in the same location or performs the action `close`, and proceeds to location `s5` then performs the flow action `startID01_4` for starting TA06, which captures the translation of the process `STOP` (deadlock).

Translating External Choice

Similarly, in translating external choice, we provide additional transitions that enable the behaviour of the chosen process to block the behaviour of the other processes. Initially, the translated behaviour makes the initials of the translated processes available such that choosing one of the processes block the other alternative processes with the co-actions of the provided additional transitions of translating external choice, as defined in Definition 3.2.

For instance, consider a process `P = P1[]P2` that composes `P1` and `P2` with operator external choice such that the translated behaviour of the process `P` is denoted by `Tp` (a list of TAs for the translation of the process `P`). In similar manner, `Tp1` and `Tp2` are lists of TA for the translation of the processes `P1` and `P2`, respectively. Then, the first TA in the list `Tp1` has additional transitions for the initials of `P2` such that choosing the behaviour `Tp2` block the alternative behaviour `Tp1`. Similarly, the first TA in the list of TA `Tp2` has additional transitions for the initials of `P1` such that choosing the behaviour `Tp1` blocks the behaviour `Tp2`. Additional examples will follow later in this chapter and Appendix D.



(a) TA01 (b) TA02 (c) TA03 (d) TA04 (e) TA05

Figure 25: A list of TA for the translated behaviour of the process Pe.

An example of translating external choice is provided in Figure 25 for the process `Pe = (left->STOP)[](right->STOP))`, which composes the two processes `(left->STOP)` and `(right->STOP)` with the operator of external choice. In Figure 25, starting from the top-left, the first TA01 is a translation of the operator external choice. TA02 and TA04 are translations of the LHS process `left->SKIP`. Then, TA03 and TA05 capture the translation of the RHS process `(right->SKIP)`. From the beginning, TA01 has three transitions, each labelled with a flow action. The TA begins with the first flow action `startIDpExtChoice?`, then starts the two TAs, for the translation of both the left and right process, using the flow actions `startID00_1!` and `startId01_2!`, respectively.

Second, TA02 is a translation the event `left`. Initially, the TA synchronises on the flow action `startID00_1` and moves to location `s2` where the TA has 3 possible transitions labelled with the actions: `left_exch?`, `right_exch!` and `tock?`. With the co-action `tock?`, the TA records the progress of time and remains on the same location `s2`. With the co-action `right_exch?`, the TA performs an external choice co-action for blocking the TA of the LHS process when the environment chooses the right process, and the TA returns to its initial location `s1`. Lastly, the TA performs the action `left_exch!` when the environment chooses the LHS process, and the TA progress to location `s3` to perform the chosen action `left` that leads to location `s5` for performing the flow action `startID00_2`, which activates TA03 for the subsequent process `STOP`. This describes the behaviour of the LHS process `left->STOP`.

Fourth, TA04 is a translation of `right`, similar to the previous translation of `left` in the second TA. Fifth, TA05 is a translation of the process `STOP`. The omitted environment TA is similar to the last TA in Figure 23.

Translating Hiding and Renaming

Also, in *tock-CSP*, an event can be renamed or hidden from the environment. In handling renaming, the translation technique carries a list of renamed events. Before translating each event, the technique checks if the event is part of the renamed events, and then translates the event appropriately with the corresponding new name. In the same manner, if an event is part of the hidden events, the technique stores a list of hidden events, such that on translating a hidden event the technique uses a special name *itau* [11] in place of the hidden event.

Later in this section, we provide a function `transform()`, which composes the three functions (`transTA()`, `envTA()` and `syncTA()`) into a system. Details description of these three functions will come later in this section. The function `transform()` prepares the required arguments of the three functions and collects the output list of TA from these three functions, and then the function `transform()` assembles the list of TA into an XML file that is suitable for Uppaal toolbox.

This completes the description of the strategy we follow in developing the translation technique. The following section describes the details of the translation rules, and provides examples that demonstrate using each rule in translating a *tock-CSP* process.

3.3. Translation Rules

This section discusses the details of the translation rules. The section describes the translation rules in a functional style. Then, the section proceeds with presenting each translation rule separately. In addition, we provide examples that illustrate using each of the translation rules in translating a *tock-CSP* process.

Example 3.3. Here, we use a simple example of TA to illustrate the definition of TA provided in Figure 26, which is defined in Listing 1 using the syntax of Haskell for expressing the translation work. The TA has two locations A and B with an inner circle in the initial location A.

[11] A special event similar to the event tau in FDR to represent hidden event.

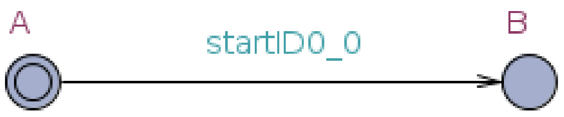


Figure 26: A sample output TA with two locations and one transition.

Line 1 defines a TA with 6 arguments. First parameter `"idTA"` specifies an identifier for the TA, similar to the naming format we use in the translation work. Second and third parameters are empty lists for both parameters [12] of the TA itself and its local definitions. Fourth, `[loc1, loc2]` is a list of locations for the TA that contains 2 locations, `loc1` and `loc2`. Fifth, `(init loc1)` specifies loc1 as the initial location of the TA. Lastly, `[tran1]` is a list of transitions that has one transition for the TA.

```
1  TA  idTA   []  []  [loc1, loc2] (Init loc1) [tran1]
2    where
3      idTA   = "ta1" + 0   + "_" + 0
4        --   = Location   ID     Name   Label        LocType
5      loc1   = Location "idA"   "A"    EmptyLabel   None
6      loc2   = Location "idB"   "B"    EmptyLabel   None
7
8         -- = Transition   Source   Target   [Label]
9      tran1 = Transition   loc1     loc2     [lab1]
10
11     lab1   = Sync (VariableID  "start")   Excl
```

Listing 1: An abstract definition of a TA that has two locations and one transition.

Line 3 highlights a definition location from Definition 3.5. Then, Line 4 defines `Loc1` as an instance of location with an identifier `"idA"` and name `"A"`, with an empty label that indicates no constraint in the location, and also specifies the type of the location to be None. In the like manner, Line 5 defines `loc2` as the second location with an identifier `"idB"`, name `"B"`, also an empty label that specifies no constraint in the location, and specifies a type for the location to be of type None.

Line 8 is a comment that highlights a definition of a transition from Definition 3.6. Then, Line 9 defines `tran1` as a transition that connects two locations `loc1` and `loc2` with `[lab1]` as a label of the transition. `lab1` is defined in Line 11 using the definition of label from Definition 3.6 as an Uppaal action with identifier "start" that has direction Excl which specifies the action as a sender.

The above Example 3.3 illustrates a simple form of the output TA produced by the translation function `transTA`. However, in the translation rule, we will have a TA that has more than two locations and multiple transitions. The upcoming translation rules define the function `transTA`. Each rule defines a translation of one of the constructors

[12] A TA has its own local parameters for expressing its behaviour.

of the BNF previously presented in Section 3.1. In the next section, we discuss details of each of the translation rules together with an example for illustrating using the rule in translating a process.

Definition 3.5. Location

```
data Location = Location ID Name Label LocType
```

Definition 3.5 defines a location with a constructor that has 4 parameters, of types `ID, Name, Label and LocType`. First parameter of type `ID` is an identifier for the location. Second parameter of type `Name` is a tag for the location. Third parameter of type `Label` is a constraint label for the location, defined below in Definition 3.7. Last parameter of type `LocType` is a format of the location, which can be one of these three: `urgent`, `committed`, `None` (which means neither urgent nor committed, just normal location with no constraint).

Definition 3.6. Transition

```
data Transition = Transition Source Target [Label]
```

Also, Definition 3.6 defines a data type for Transition, which has a constructor with 3 parameters, of type `Source`, `Target` and `[Label]`. First parameter of type `Source`, is a starting location for the transition. The second parameter of type `Target` is a destination location for the transition. The third parameter of type `[Label]` is a list of labels for the transition.

Definition 3.7. Label

```
data Label = EmptyLabel
           | Invariant     Expression
           | Guard         Expression
           | Update       [Expression]
           | Sync          Identifier   Direction
```

Finally, the label is an expression (or list of expressions) that is associated with either a location or a transition. For a location, a label can be empty or invariant that specifies the constraint condition of the location. While for a transition the label can be either empty for silent transition or any combination of these three types: `Guard`, `Update` and `Sync`. Where `Sync` is a type for an Uppaal action that has an identifier and direction, which is either sender (with a question mark) or receiver (with an exclamation mark).

Definition 3.8. Function `transTA`

```
transTA :: CSPproc -> ProcName  -> BranchID -> StartID ->
    FinishID -> UsedNames -> ([TA], [Event], [SyncPoint])
```

The function `transTA` has 6 parameters. The type of the parameters are `CSPproc`, `ProcName`, `BranchID`, `StartID`, `FinishID` and `UsedNames`. The first parameter of type `CSPproc`, is the input *tock-CSP* process to be translated. The second parameter is a name for the process, of type `ProcName`; an alias for `String`. While third and fourth parameters are of type `BranchID` and `StartID`; also the alias of type String and Int respectively. We use the two parameters to generate an identifier for each small TA in the list of the output TA. In generating the identifiers, we consider the structure of the binary tree for the AST of the input process, which has branch and depth. So, a combination of these two parameters branch and depth identifies the position of each TA in the list of the output TA. Fifth parameter of type `FinishID`, is a termination ID. The last parameter of type `UsedNames` is a collection of names, which we used in defining the translation function, mainly for passing translation information from one recursive call to another. We will explain the purpose of these names as we introduce them in the translation rules. The first three parameters `ProcName`, `BranchID` and `StartID` are essential for each translation rule.

3.3.1. Translation of STOP

This section describes a translation of a constant process `STOP`. The section begins with presenting a rule for translating STOP and then follows with an example that illustrates using the rule in translating a process.

Rule 3.1. Translation of STOP

```
1  transTA STOP processName bid sid _ _ =
2        (([(TA idTA [] [] locs [] (Init loc1) trans)]), [], [] )
3        where
4           idTA = "taSTOP__" ++ bid ++ show sid
5
6           --   = Location ID      Name    Label       LocType
7           loc1 = Location "id1"   "s1"    EmptyLabel  None
8           loc2 = Location "id2"   "s2"    EmptyLabel  None
9           locs = [loc1, loc2]
10
11          --    = Transition Source   Target  [Label] [Edge]
12          tran1 = Transition loc1     loc2    [lab1]  []
13          tran2 = Transition loc2     loc2    [lab2]  []
14          intrp = transIntrpt intrptsInits loc1 loc2
15          trans = [tran1, tran2] ++ intrp
16
17          lab1 = Sync (VariableID
18                        (startEvent processName (bid ++ sid)) [])
19                      Ques
20          lab2 = Sync (VariableID "tock" []) Ques
21
22          -- Get initial events for possible interrupting process
23          (_, _, _, _, _, intrptsInits, _, _) = usedNames
```

Rule 3.1 expresses the translation of the construct STOP, which produces an output TA depicted in Figure 27. The figure illustrates the structure of the output TA that has two locations and two transitions as defined in Lines 7–9 and Lines 11–13, respectively.

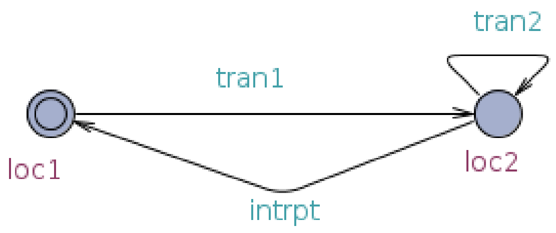


Figure 27: A structure of TA for the translation of STOP.

Starting from the beginning of the translation rule, Line 1 provides a definition of the function `transTA` for the construct `STOP` and the 3 essential parameters for translating the construct STOP, `processName`, `bid` and `sid` . While the remaining two underscores represent unused arguments for this translation rule. In Haskell, an underscore indicates a position of unused arguments. For conciseness, we use the underscore to omit unused arguments and provide only the required arguments for each translation rule.

Line 2 defines the output tuple which contains three elements, a list of output TA, and the remaining two elements for translating multi-synchronisation. For this translation rule, there is no multi-synchronisation, so the remaining two elements are both empty for the synchronisation actions and their corresponding identifiers.

Also, in the output tuples, the first element (non-empty element) is a definition of the output TA for the translation of the constant process STOP, which has six parameters. First, `idTA` is an identifier for the TA, which is defined subsequently in Line 4, as concatenation of the keyword `"taSTOP___"` with the 2nd and 3rd arguments of the function `transTA`, that is `bid` and `sid` respectively. Additionally, still in Line 2, in the definition of the output TA, the 2nd, 3rd and 5th parameters are empty for the output TA. While the 4th parameter `locs` is a list of locations for the output TA defined in Lines 7–9. The 6th parameter `(Init loc1)` specifies `loc1` as the initial location of the output TA. Lastly, `trans` describes a list of transitions that connect the two locations as defined in Line 12–13. Lines 14 defines interrupt transitions in the case of translating a process that is composed with an operator interrupt.

Finally, Lines 17 – 20 define the labels of the two transitions in the output TA. The first label `lab1` defines a label for the first transition as a first flow action, which we generate its name using the function `startEvent`, as defined in the following Definition 3.9. The second label `Lab2` for the second transition is defined as an Uppaal action "tock" with `Ques`, which indicates a receiving action.

Definition 3.9. Function `startEvent`

```
startEvent :: String      -> String -> Int       -> String
startEvent    processName    bid       sid       =
                             if    notNull       processName
                             then "startID" ++  processName
                             else "startID" ++  bid ++ show sid
```

The function `startEvent` generates the name of the first flow action. If the input process has an identifier, we use the identifier in the translated TA as the name of the first flow action, else the function generates a new name for the first flow action. The name is a combination of the keyword `"startID"` with the two identifiers of the first TA in the list of the translated output, that is a combination of the parameters `BranchID` and `StartID`.

Declaration of the function `transIntrpt`

```
transIntrpt :: [Event] -> Location -> Location -> [Transition]
```

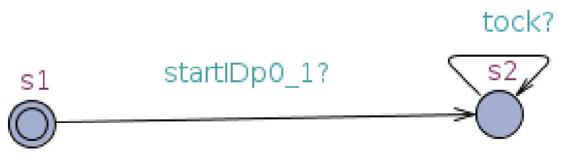
Line 14 defines interrupt transitions using the initials of an interrupting process defined in the function `transIntrpt`, as highlighted in the translation strategy in Section 3.2. The function `transIntrpt` has three parameters. The types of the parameters are the list of events (initials of an interrupting process) and two locations that connect the transition for interrupt. The first argument `intrpts` is the initials of an interrupting process and generates a transition for each of the initials.

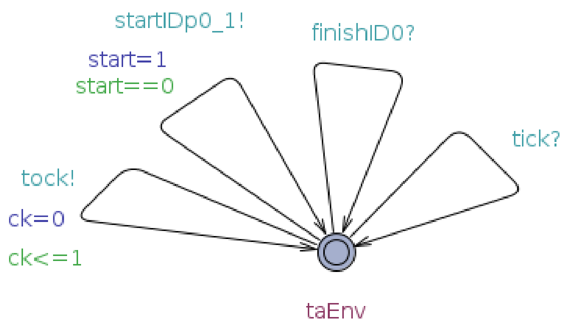
The list of the initials `intrpts` comes from the tuple `usedNames` (Line 23). Previously, we mentioned that we would explain the names in the point where we start using the names. Here, we start using the name `intrpts` from the names `usedNames`. The name `intrpts` is used to collect the initials of interrupting processes for constructing interrupting transitions that enable a translated process to interrupt the behaviour of another process.

The behaviour of the output TA begins with the first flow action (line 17), which is constructed using a function `startEvent`, previously defined in Definition 3.9. After that, the TA performs the action `tock` (line 18), repeatedly, which allows time to progress. An illustration of using this translation rule is provided in the following Example 3.4.

Example 3.4. An example of translating a process STOP produces a list of TA that contains two TAs, as illustrated below.

```
transTA STOP "p0_1" 1 0 0 ([], [], [], [], [], [], [], ([],[]))
     =  [
```





```
        ]
```

Example 3.4 demonstrates a translation of the process STOP using Rule 3.1, which produces a list of translated TA that contains two TA, a small TA and its corresponding environment TA. The behaviour of the output TA begins with the environment TA that performs its first flow action `startIDp0_1!` with the cooperation of the small

TA using its corresponding co-action `startIDp0_1?`. Then, the small TA continues performing the event `tock` for the progress of time, and remains in location `s2`. This concludes the behaviour of the translated TA for the constant process `STOP`.

3.3.2. Translation of Stopu (Timelock)

This section describes a translation of constant process `Stopu`, an urgent deadlock that does not allow time to pass. The section begins with presenting a rule for translating the process `Stopu`. Then, follows with an example that illustrates using the rule in translating a process.

Rule 3.2 expresses the translation of the constant process `Stopu` that produces an output TA that is depicted in the following Figure 28, which is annotated with the names used in the translation rule. The figure illustrates the structure of the output TA, which has two locations and one transition as defined in Lines 7–9 and Line 12 respectively.

This description of Rule 3.2 resembles the previous description of Rule 3.1 (translation of STOP), except that the output TA of this rule does not perform the event `tock` that allows time to progress in the previous rule. The structure of this output TA has two locations, `loc1` and `loc2` as defined in Lines 7 and 8, respectively, and only one transition for the coordinating start event (Line 11). This behaviour of the output TA begins with synchronising on the first coordinating start event and then deadlock immediately. This is illustrated in the following example for translating the constant process `Stopu`.

Rule 3.2. Translation of Stop*u*

```
1  transTA Stopu processName bid sid _ _ =
2      ([(TA idTA [] [] locs [] (Init loc1) trans)], [], [])
3      where
4        idTA = ("taSTOP___" ++ bid ++ show sid)
5
6        --  = Location   ID     Name Label       LocType
7        loc1 = Location "id1" "s1"  EmptyLabel  None
8        loc2 = Location "id2" "s2"  EmptyLabel  None
9        locs = [loc1, loc2]
10
11       --   =  Transition  Source   Target   [Label] [Edge]
12       trans = [Transition  loc1     loc2     [lab1]  []   ]
13
14       lab1  = Sync (VariableID
15                        (startEvent processName (bid ++ sid)) [])
16                     Ques
```

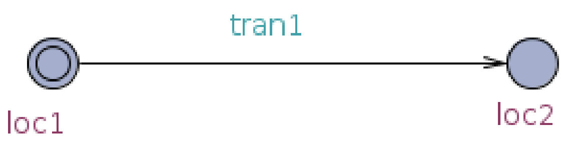
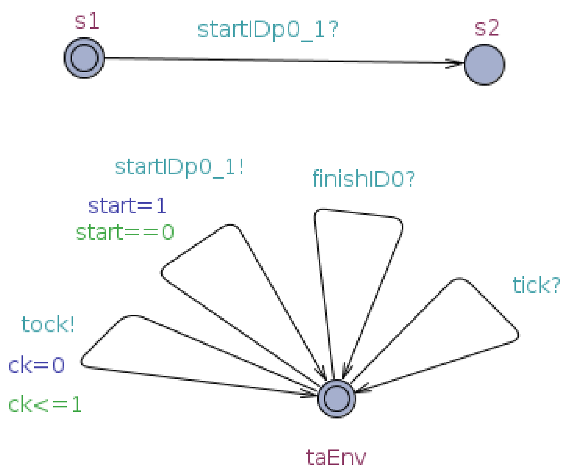


Figure 28: A structure of TA for the translation of `Stopu`

Example 3.5. An example for translating an urgent process Stopu.

```
-- transTA :: CSPproc -> procName -> BranchID  -> StartID
--                    -> FinishID -> UsedNames ->
--                       ([TA], [Event], [SyncPoint])

transTA Stopu "p0_1" "0" 1 0
        ([], [], [], [], [], [], [], ([],[]))
    = [
```



```
        ]
```

The above Example 3.7 illustrates a translation of the constant process `Stopu` according to Rule 3.2. Also, this example resembles the previous Example 3.4, except that the behaviour of this TA terminates immediately without performing the event `tock`. In this example, the output TA synchronises on the coordinating start event `startID00` and then deadlocks immediately.

3.3.3. Translation of SKIP

This section describes the translation of process SKIP. The section begins with presenting a rule for translating the process SKIP. Then, follows with an example that illustrates using the rule in translating a process. However, the behaviour of SKIP can be interrupted just before termination, so we have to consider a possible interrupt in translating interrupt.

Rule 3.3 describes a translation of process `SKIP` into a single output TA, which is depicted in the following Figure 29. The figure is annotated with the names used in the translation rule. The structure of the output TA has 3 locations: `loc1`, `loc2` and `loc3` (define in Lines 6 – 9) and 4 transitions `tran1`, `tran2` and `tran3` (define in Lines 11 – 16).

The behaviour of the TA begins on `tran1` for a flow action that is defined using the function `startEvent` (Definition 3.9). Then, the TA follows one of the 3 transitions: `tran2, tran3 or intrp`. On transition `tran2`, the output TA performs the event `tock` to record the progress of time, and remains in the same location `loc2`. On transition `tran3`, the output TA performs the event `tick` and then immediately follows the subsequent transition `tran4` to perform the termination event `finishID0!`. Finally, on transition `intrp`, the TA is interrupted by another process.

Line 15 defines an interrupt transition, which is provided for the case of translating a process that involves interrupt. Details of translating interrupt will be provided in Section 3.3.13. Here, we highlight a declaration of the function `transIntrpt` below, due to its first appearance in this translation rule.

Rule 3.3. Translation of SKIP

```
1  transTA SKIP procName bid sid fid usedNames =
2       ([(TA idTA [] [] locs [] (Init loc1) trans)], [], [])
3       where
4          idTA = "taWait_n" ++ bid ++ show sid
5          loc1 = Location "id1" "s1" EmptyLabel None
6          loc2 = Location "id2" "s2" EmptyLabel None
7          loc3 = Location "id3" "s3" EmptyLabel CommittedLoc
8          locs = [loc1, loc2, loc3]
9
10         tran1 = Transition loc1 loc2 [lab1]  []
11         tran2 = Transition loc2 loc2 [lab2]  []
12         tran3 = Transition loc2 loc3 [lab3]  []
13         tran4 = Transition loc3 loc1 [lab4]  []
14         intrp = transIntrpt intrptsInits loc1 loc2
15         trans = [tran1, tran2, tran3, tran4] ++ intrpt
16
17         lab1 = Sync (VariableID (startEvent processName
18                     (bid ++ sid)) []) Ques
19         lab2 = Sync (VariableID "tock" []) Ques
20         lab3 = Sync (VariableID "tick" []) Excl
21         lab4 = Sync (VariableID  finishLab []) Excl
22
23         finishLab = ("finishID" ++ show fid)
24
25         -- Gets initial events of an interrupting process
26         (_, _, _, _, _, intrptsInits, _, _) = usedNames
```

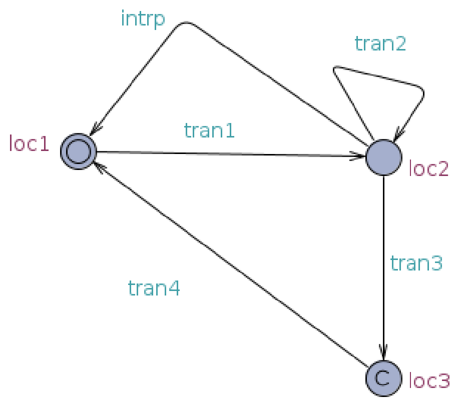
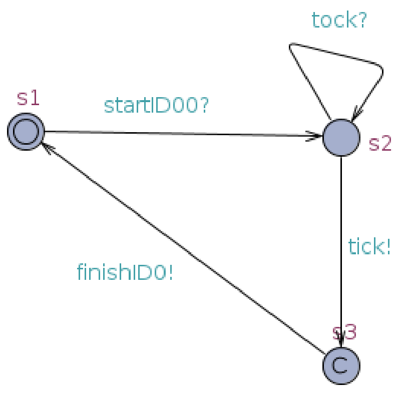


Figure 29: A structure of a TA for the translation of the process SKIP.

The list of the initials `intrpts` comes from the tuple `usedNames` (Line 27). Previously, we mentioned that we would explain the names in the point where we start using the names. In this rule (Line 27), we start using the name `intrpts` from the names `usedNames`. We used the name `intrpts` to collect the initials of interrupting processes for constructing interrupting transitions. This completes the description of Rule 3.3. The following Example 3.6 illustrate using the rule for translating constant process `SKIP`.

Example 3.6. An example for translating a constant process `SKIP`.

```
transTA SKIP "" "0" 0 0
        ([], [], [], [], [], [], [], ([],[])) =
        [
```



```
        ]
```

Example 3.6 illustrates using Rule 3.6 in a translating a constant process `SKIP`. The example uses the definition of the translation function `transTA` for the construct `SKIP` and the required parameters: process `SKIP`, empty name, ”0” for `BranchID`, 0 for `StartID`, 0 for `finishID`, and a tuple of empty lists for the `usedNames`. We used an empty name to illustrate the translation of a process that has an empty name.

Details of the output TA are as follows. Initially, the output TA synchronises on the start event `startID00?`. In this example, the translated process does not have a name, so the start event is a concatenation of the keyword ”startID” with the number 0 for the parameter `BranchID` and another number 0 for the parameter `StartID`. After that, on location `s2` either the TA performs the event `tock` and returns to the same location; or performs the event `tick` and then immediately performs the termination event `finishID0`, which notifies the TA environment for a successful termination of the output TA.

In this example, there is no interrupting process, so the function `transIntrpt` produces an empty list for the interrupting transitions. Section 3.3.13 provides an example that has interrupting transitions. For better understanding, we will discuss the example with interrupt transitions after discussing the translation of the construct `interrupt`. This completes the description of an example for translating a constant process `SKIP`.

3.3.4. Translation of Skipu (Urgent termination)

This section describes the translation of another constant process `Skipu`, which specifies an urgent termination that does not allow time to pass before the termination. The section begins with presenting a rule for translating the process `Skipu`. And then, follows with an example that illustrates using the rule in translating a process.

Rule 3.4. Translation of Skipu

```
transTA Skipu procName bid sid fid usedNames =
      ([(TA idTA [] [] locs [] (Init loc1) trans)], [], [])
      where
        idTA = "taSkipu_" ++ bid ++ show sid

        loc1 = Location "id1" "s1" EmptyLabel None
        loc2 = Location "id2" "s2" EmptyLabel None
        loc3 = Location "id3" "s3" EmptyLabel CommittedLoc
        locs = [loc1, loc2, loc3]

        tran1 = Transition loc1 loc2 [lab1]  [] tran3 =
           Transition loc2 loc3 [lab3]  []
        tran4 = Transition loc3 loc1 [lab4]  [] trans = [tran1,
           tran3, tran4]

        lab1 = Sync (VariableID (startEvent procName bid sid)
           [])
                    Ques

        lab3 = Sync (VariableID "tick"         []) Excl
        lab4 = Sync (VariableID ("finishID" ++ show fid) [])
           Excl
```

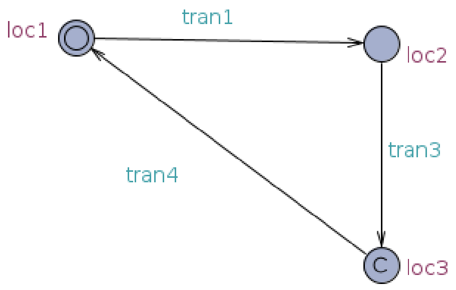


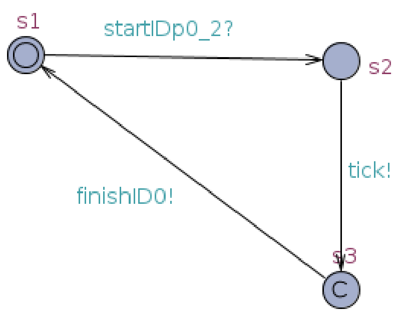
Figure 30: A structure of TA for the translation of urgent termination.

Rule 3.4 resembles the previous Rule 3.3, except that on the location `s2` the output

TA does not perform the event `tock`. This means that the TA terminates immediately, as illustrated in Figure 30. The following example demonstrates using the rule in translating a process.

Example 3.7. An example for translating a process for an immediate termination.

```
transTA Skipu "" "0" 0 0
         ([], [], [], [], [], [], [], ([],[]))
         = [
```



```
          ]
```

Similarly, Example 3.7 resembles Example 3.6, except that the output TA terminates immediately. Initially, the TA synchronises on the coordinating start action `startIDp0_2`, then the TA performes the event `tick` and proceeds immediately to perform a termination action `finishID0`, which indicates successful termination. There is no interrupting process in this example, so the interrupting transitions are empty. This completes the description of Example 3.7, which illustrates a translation of the constant process `Skipu` for urgent termination.

3.3.5. Translation of Prefix

This section describes the translation of operator `Prefix`. The section begins with presenting a rule for translating the operator `Prefix` and then follows with an example that illustrates using the rule in translating a process.

This rule for translating the operator prefix happens to be the largest translation rule because in translating each event, we need to check if the event is part of events that are hidden, renamed, synchronisation, initial of external choice or initial of another interrupting process. In each of these cases, the TA has different behaviour that has to be translated according to the specification of the process.

First, the translation rule defines a function for checking both hidden and renamed events and then defines eight cases for capturing the possible behaviour of an event in a process that is part of synchronisation, external choice or interrupt. Due to the length of the rule, here we provide a short version of the rule with a complete definition of one case. Complete details of the rule are available in Appendix A.

Rule 3.5. Translation of Prefix

```
transTA (Prefix e1 p) procName bid sid fid usedNames =
        (([(TA idTA [] [] locs1 [] (Init loc1) trans1)] ++ ta1),
                sync1, syncMapUpdate)
  where
    idTA = "taPrefix" ++ bid ++ show sid
    (syncs, syncMaps, hides, renames, exChs, intrpts,
       initIntrpts, excps) = usedNames

    -- Checking hiding or renaming
    e = checkHidingAndRenaming e1 hides renames

    -- High level definition of locations and transitions for
    -- the eight possible combination of synchronisation, choice
    -- and interrupt, 000, 001, 010, 011, 100, 101, 110, 111
    (locs1, trans1)
      |((not synch) && (not exChoice) && (not interupt)) = case1
      |((not synch) && (not exChoice) && (    interupt)) = case2
      |((not synch) && (    exChoice) && (not interupt)) = case3
      |((not synch) && (    exChoice) && (    interupt)) = case4
      |((    synch) && (not exChoice) && (not interupt)) = case5
      |((    synch) && (not exChoice) && (    interupt)) = case6
      |((    synch) && (    exChoice) && (not interupt)) = case7
      |((    synch) && (    exChoice) && (    interupt)) = case8

    case1 = ([loc1, loc2, loc5], [t12, t22, t25,  t51])
    loc1  =  Location "id1"  "s1"  EmptyLabel None
    loc2  =  Location "id2"  "s2"  EmptyLabel None
    loc5  =  Location "id5"  "s5"  EmptyLabel CommittedLoc
    t12   =  Transition  loc1  loc2   lab1    []
    t25   =  Transition  loc2  loc5   lab2    []
    t51   =  Transition  loc5  loc1   lab3    []
    t22   =  Transition  loc2  loc2   labTock []

    lab1 = [Sync (VariableID (startEvent procName bid sid) [])
       Ques]
    lab2 = [Sync (VariableID (show e) []) Excl]
    lab3 = [Sync (VariableID ("startID" ++ bid ++ "_" ++
                      show (sid+1)) []) Excl]
    labTock = [Sync (VariableID  "tock" [])  Ques]
```

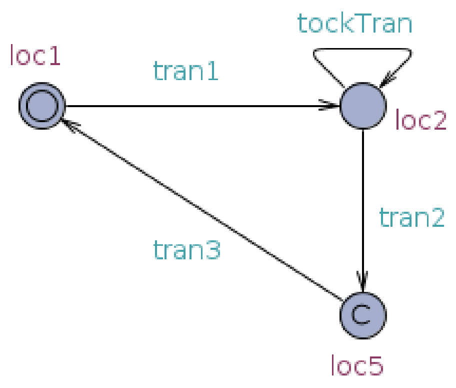


Figure 31: A structure of TA for the translation of Prefix.

As discussed in Section 3.1, the operator prefix is a binary operator that combines an event with a process, syntactically in the form of `event->Process`. The prefix event is translated according to one of the eight possible cases for a process that takes part in synchronisation, external choice and interrupt. Each case defines a different behaviour for the prefix event in the translation rule.

In Figure 31, we annotate the structure of the TA for translation of an event in case 1. The TA has 3 locations and four transitions, as defined in Lines 27–29 and Lines 31–37 respectively. The TA begins with transition `tran1` for performing flow action, and then on location `loc2` either the TA performs the action `tock` to record the progress of time and return to the same location, or perform the translated action on transition `tran2` that leads to location `loc3`. Then, on transition `tran2`, the TA performs another flow action to activate the subsequent TA. The remaining 7 cases follow a similar pattern. Details of the remaining 7 cases are provided in Appendix A.

Cases 1 to 4 are cases that did not involve synchronisations. Case 1 is the simple case where a prefix event is not part of any of one of the three operators. Case 2 defines a translation of an event that is part of the initials of an interrupting process, which means that the event is the kind of event that interrupts the behaviour of another process. Case 3 is for an event that is part of the initials of a process that participate in external choice only. Case 4 is for an event that is part of a process that engages in both external choice and interruption.

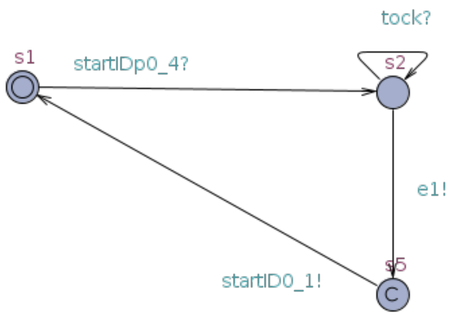
Cases 5 to 8 are cases that involve synchronisations. Case 5 defines a translation of an event that is part of synchronisation only, which means that multiple processes synchronise on performing the event. Case 6 is for the translation of an event that is part of both synchronisations and interrupts. Case 7 is for the translation of an event that is part of both synchronisation events and the initials of external choice events. Finally, case 8 defines a translation of an event that is part of the three operators: synchronisation, external choice and interrupt.

For each of these 8 cases, Rule 3.5 defines a separate TA for translating the behaviour of a prefix event. Definition of all the locations and transitions of these eight possible TAs generates a long list of definitions that makes the size of the rule very large.

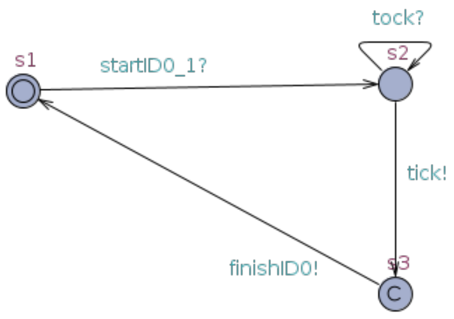
Here, we present a high-level definition of the rule for the main part, which omits the detailed description of locations and transitions of all the possible output TA for this translation rule. Details of the translation rule with the complete definitions for all the locations and transitions are available in Appendix A. In Figure 31, we map the names with the structure of TA defined in case 1 of the translation rule. Example 3.8 illustrates using the translation Rule 3.5 in translating a process.

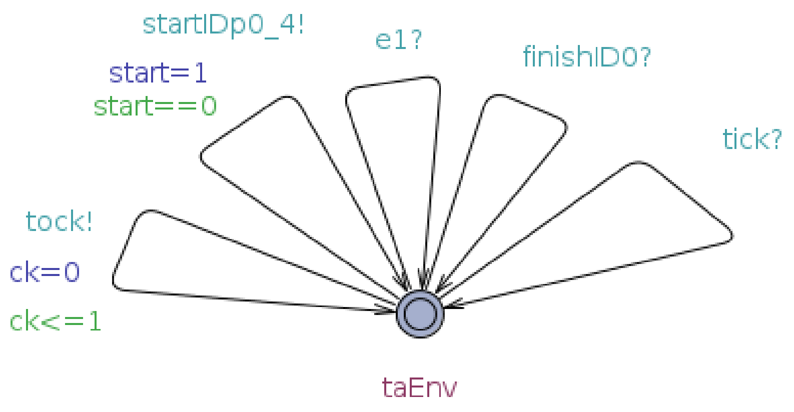
Example 3.8. An example that demonstrates using Rule 3.5 in translating a process `e1->SKIP`

```
transTA e1->SKIP "p04" "0" 0 0 usedNames =
    [
```



```
    ] ++ transTA(SKIP)
   = [
```





```
        ]
```

Example 3.8 illustrates using Rule 3.5 in translating a process `e1->SKIP`, which is translated into a list containing three TA. The first Ta captures the translation of the event `e1` using Rule 3.5. The second TA captures the translation of the subsequent process `SKIP` using Rule 3.3. The last TA is an environment TA for the list of the translated TA.

The detailed behaviour of the output TA is as follows. Initially, the first TA synchronises on the coordination action `startIDp04`. Then, on location `s2` either the TA performs the event `tock` and remains in the same location `s2` or the TA performs the prefix event `e1` that leads to performing the subsequent flow action `startID01` to activate to activate the second TA, which synchronises on the flow action `startID01`. Then, either the second TA performs the action `tock` and remains in the same location; or `TA1` performs the action `tick` which leads to performing the termination action `finishID` for a successful termination. These two TA describe the translation of process `e1->SKIP`.

3.3.6. Translation of WAIT n

This section describes a translation of process `(WAIT n)`, which defines a delay of at least `n` units time. The section begins with presenting a rule for translating the process `(WAIT n)`, and then follows with an example that illustrates using the rule in translating a process.

Rule 3.6. Translation of WAIT n

```
transTA (WAIT 0) processName bid sid fid usedNames =
    transTA SKIP processName bid sid fid usedNames

transTA (WAIT n) processName bid sid fid usedNames =
    transTA (Prefix Tock Wait (n - 1)) [] bid sid fid usedNames
```

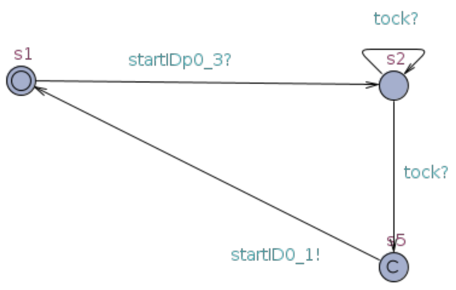
Rule 3.6 describes a translation of delay process `WAIT(n)`, which is translated in terms of two constructs: `Prefix` and `SKIP`, previously defined in Rule 3.3 and Rule 3.5, respectively. In the syntax of *tock-CSP*, this is expressed as:

```
Wait(n) = tock -> Wait (n-1).
```

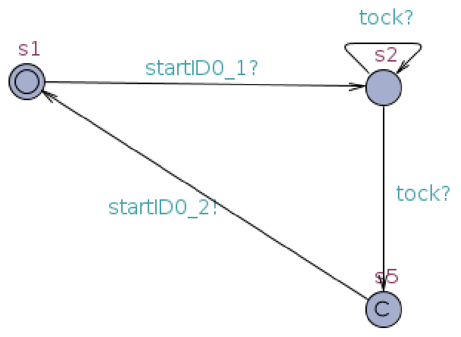
The process `WAIT(n)` is translated into a list of TA, which performs the event `tock` n times until the value `n` becomes `0` and the the TA behaves as `SKIP`. The base case is translated according to Rule 3.6. While the remaining cases are translated according to Rule 3.5. The following example illustrates using the rule in translating a process.

Example 3.9. An example for translating a delay process `WAIT(2)`, which expresses a delay of 2 units time. The process is translated as follows.

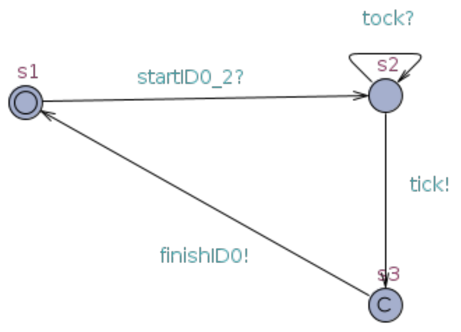
```
transTA (WAIT 2) "p0_3" "0" 0 0
        ([], [], [], [], [], [], [], ([],[])) =
        [
```

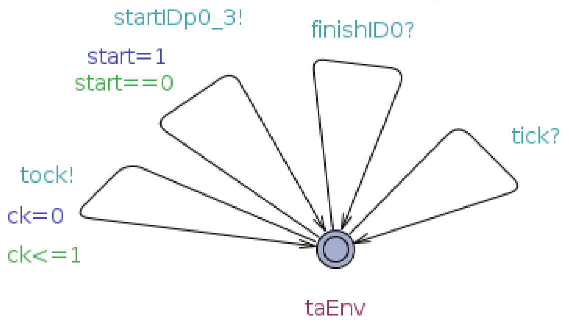


```
        ] ++
        transTA (WAIT 1) "" "0" 1 0
                ([], [], [], [], [], [], [], ([],[])) = [
```



```
          ] ++
          transTA (WAIT 0) "" "0" 2 0
                  ([], [], [], [], [], [], [], ([],[])) =
```





```
          ]
```

Example 3.9 illustrates using Rule 3.6 in translating the process `WAIT(2)`. Initially, the example defines the function `transTA` for the construct `(WAIT n)` and its required arguments: process is `WAIT(2)`, process name is ”p0_3”, `branchID` is ”0”, `startID` is 0, `finishID` is 0 and `usedNames` is the remaining empty lists for the collection of names that are empty at the beginning. The translation produces a list of TA `TA0, TA1 and TA2` in the example.

Details behaviour of the output TA is as follows. First, `TA0` synchronises on the flow action `startIDp0_3`, which connects the environment with the first TA in the list of the translated TA. Then, on location `s2`, `TA0` performs the time event `tock` at least once and then performs the subsequent flow action `startID01`, which connects two

TA0 and TA1. In this case, TA1 is similar to the previous TA0 because both of them capture the translation of the event tock; TA1 performs the second action tock and then performs another flow action startID02, which connects TA1 and TA2. Lastly, TA2 synchronises on the flow action startID02 and then either TA2 performs the action tock and remains in the same location; or TA2 performs the action tick and then immediately proceeds to a terminating action finishID0, which indicates a successful termination of the translated process. These three TAs describe the translation of the process WAIT(2).

3.3.7. Translation of Waitu n (Strict delay)

This section describes the translation of process Waitu n, a strict delay of n time units. The section begins with presenting a rule for translating the process Waitu n and then follows with an example that illustrates using the rule in translating a process.

Rule 3.7 describes a translation of strict delay. In Figure 32, we annotate the structure of the output TA with the names used in the translation rule. The structure of the TA has 2 locations and three transitions as defined in Lines 6–8 and Lines 10–14. Then, Line 16 extracts the used names for interrupt. And Lines 18–20 defines the labels of the transitions. Lines 22–27 defines the guards for controlling the deadlines. Finally, Lines 30–31 reset the deadline in case of a translating process that has recursive calls.

The behaviour of the output TA begins on transition tran1, where the TA synchronises on a flow action (defined in Line 18). Then, on Location loc2 (defined in Line 6), either TA follows transition tran2 or tran3. On transition tran2 (define in Line 10), the TA checks the delay guard dlguard (defined in Line 22), if it is true the TA performs the time event tock and update the delay timer with the expression dlupdate (Lines 23–24). Alternatively, if the guard dlguard is false, the second guard dlguard2 becomes true (Line 27), which enables the TA to perform the next flow action on transition tran3, as well as resetting the timer in the expression t_reset (Lines 30–31). This transition completes the behaviour of the translated TA. The following Example 3.10 illustrates using the rule in translating a process.

Rule 3.7. Translation of Waitu n

```
transTA (Waitu n) procName bid sid fid usedNames =
  ([(TA idTA [] [] locs [] (Init loc1) trans)], [], [])

  where
    idTA = "taWait_u" ++ bid ++ show sid

    loc1 = Location "id1" "s1" EmptyLabel None
    loc2 = Location "id2" "s2" EmptyLabel None
    locs = [loc1, loc2]

    tran1 = Transition loc1 loc2 ([lab1] ++ t_reset)       []
    tran2 = Transition loc2 loc2 ([lab2] ++ dlguard
            ++ dlupdate) []
    tran3 = Transition loc2 loc1 ([lab4] ++ dlguard2
            ++ t_reset) []
    trans = [tran1, tran2, tran3] ++
            (transIntrpt intrpts loc1 loc2)

    (_, _, _, _, _, intrpts, _, _) = usedNames

    lab1 = Sync (VariableID (startEvent procName bid sid) [])
        Ques
    lab2 = Sync (VariableID "tock"  []) Ques
    lab4 = Sync (VariableID ("finishID" ++ show fid)   []) Excl

    dlguard  = [(Guard (BinaryExp (ExpID "tdeadline")
                        Lth (Val n)))]
    dlupdate = [(Update
                  [(AssgExp (ExpID "tdeadline")
                            AddAssg (Val 1))] )]

    -- A guard for exiting a strict delay
    dlguard2 = [(Guard (BinaryExp (ExpID "tdeadline")
                        Equal (Val n)))]

    -- Reset the deadline time
    t_reset = [(Update [(AssgExp (ExpID "tdeadline"
                                 ASSIGNMENT (Val 0)) ] )   ]
```

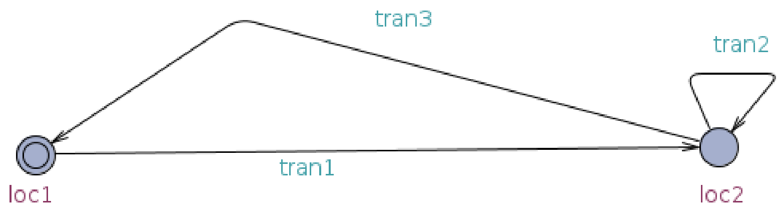
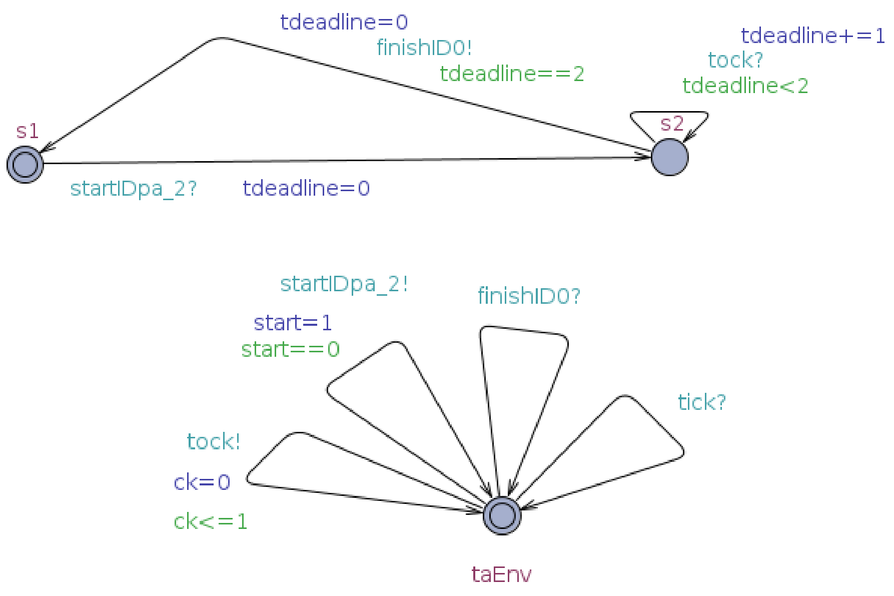


Figure 32: A structure of a TA for a translation of strict delay.

Example 3.10. An example for translating a process `Waitu(2)` a strict delay of 2 time units is illustrated in this example.

```
transTA (Waitu 2) "pa_2" "0" 0 0
          ([], [], [], [], [], [], [], ([],[]))
          ⤳ [
```



]

Example 3.10 illustrates using Rule 3.7 in translating a process `Waitu 2`. The example translates the process `Waitu 2` into a list of TA that contains two TAs. In the beginning, the example applies the function `transTA` on the required parameters: process is `Waitu 2`, process name is `pa_2`, `branchID` is ”0”, `startID` is 0, `finishID` is 0, `usedNames` is a tuple of empty elements, each rule begins with empty used names. As the translation evolves, we build a collection of the names used in the translation of a process.

In the resulting output TA, the first TA synchronises on the coordinating flow action `startIDp0_3` and then performs the action `tock` twice, which disables the first guard `(tdeadline<2)` and enables the second guard `(tdeadline==2)`. Finally, the TA performs the termination action `finishID0`. This completes the description of an example for translating the process `Waitu 2` into TA.

3.3.8. Translation of Internal Choice

This section describes a translation of operator for Internal choice. The section begins with presenting a rule for translating the operator internal choice and then follows with an example that illustrates using the rule in translating a process.

Rule 3.8. Translation of Internal Choice

```
transTA (IntChoice p1 p2) procName bid sid fid usedNames =
   ([(TA idTA [] [] locs [] (Init loc1) trans )] ++ ta1 ++ ta2,
       (sync1 ++ sync2), (syncMap1 ++ syncMap2) )
   where
      idTA  = "taIntCho" ++ bid ++ show sid
      loc1 = Location "id1" "s1" EmptyLabel None
      loc2 = Location "id2" "s2" EmptyLabel CommittedLoc
      loc3 = Location "id3" "s3" EmptyLabel CommittedLoc
      loc4 = Location "id4" "s4" EmptyLabel CommittedLoc
      locs = [loc1, loc2, loc3, loc4]

      tran1 = Transition loc1 loc2 [lab1] []
      tran2 = Transition loc2 loc3 []     []
      tran3 = Transition loc2 loc4 []     []
      tran4 = Transition loc3 loc1 [lab4] []
      tran5 = Transition loc4 loc1 [lab5] []
      trans = [tran1, tran2, tran3, tran4, tran5]

      lab1 = Sync (VariableID (startEvent procName bid sid) [])
                  Ques
      lab4 = Sync (VariableID ("startID" ++ (bid ++ "0") ++
         show (sid+1)) [])
                  Excl
      lab5 = Sync (VariableID ("startID" ++ (bid ++ "1") ++
         show (sid+2)) []) Excl

      -- translation of RHS and LHS processes
      (ta1, sync1, syncMap1) =
            transTA p1 [] (bid ++ "0") (sid+1) fid usedNames
      (ta2, sync2, syncMap2) =
            transTA p2 [] (bid ++ "1") (sid+2) fid usedNames
```

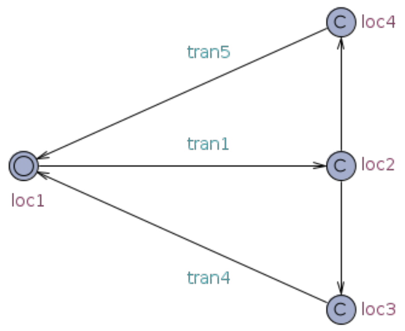


Figure 33: A structure of a TA for translating Internal choice.

Internal choice is a binary operator that combines two processes `P1` and `P2`. Rule 3.8 translates the operator of internal choice into a TA that coordinates a list of translated TA `Tp1` and `Tp2` for the translation of the two processes `P1` and `P2` respectively, that are composed with internal choice operator.

In Figure 33, we annotate the structure of the output TA with the names used in the translation rule. The output TA begins on transition `tran1` for performing a flow action that connects the TA with the network of the TA. Then, the output TA follows one of the two silent transitions that lead to transition `tran4` and `tran5` respectively. On transition `tran4` the TA activates the list of TA `Tp1` and on transition `tran5` the TA activates the list of TA `Tp2`.

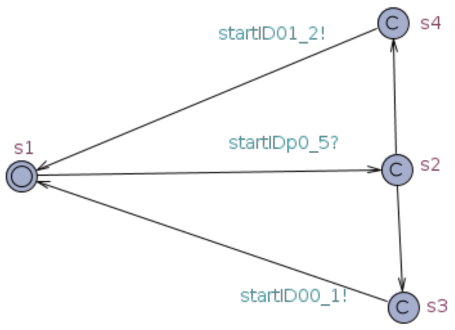
Details of Rule 3.8 are as follows. Line 1 defines the function `transTA` for the construct `IntChoice` and the 5 required parameters for this rule. Line 2 describes the output tuple that contains three elements, a list of translated TA, a list of synchronisation actions and a list of identifiers for identifying each synchronisation action.

The output TA has four locations and five transitions, as defined Lines 7–11 and Lines 13–18 respectively. Lines 20–26 define the label of the transitions. Lines 28–31 defined the subsequent translation of the processes `P1` and `P2`.

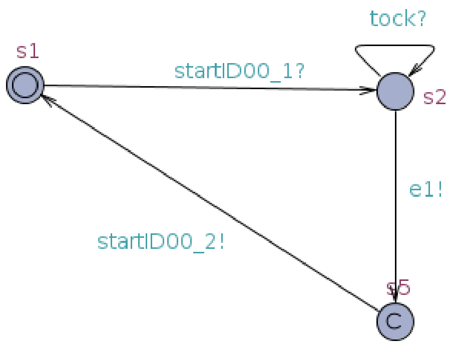
The behaviour of the output TA begins with a flow action (defined in Line 20). Then, on location `loc2`, the TA follows one of the two silent transitions, that is either `tran2` or `tran3`. Transition `tran2` leads to transition `tran4`, where the TA performs another flow action (Line 22) that activates `Tp1`. While transition `tran3` leads to transition `tran5`, where the TA performs a flow action (define Line 25) that activates `Tp2`. The following Example 3.11 illustrates using this Rule 3.8 in translating a process.

Example 3.11. An example for translating a process that composes two processes with internal choice.

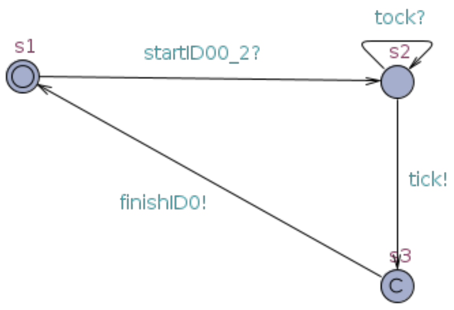
```
transTA((e1->SKIP)|-|(e2->SKIP)) =
    [
```



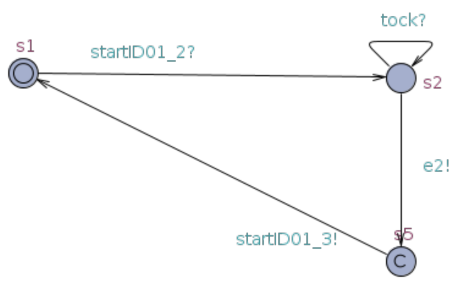
```
      ] ++ ta1 ++ ta2
where
ta1 = transTA(e1->SKIP)
    = [
```



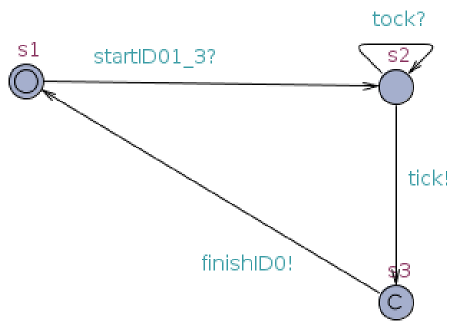
```

       ] ++ transTA(SKIP)
          = [
```

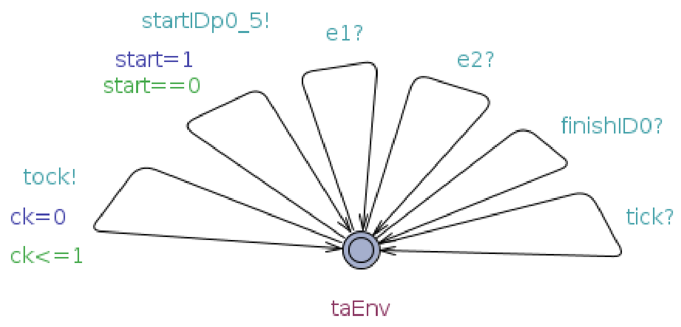


```
      ]

ta2 = transTA(e2->SKIP)
    = [
```



```
    ] ++ = transTA(SKIP)
         = [
```





```
  ]
```

Example 3.11 translates the process `((e1->SKIP)|-|(e2->SKIP))` into a list containing 5 TA. The first TA is a translation of the operator internal choice. Second and third TA are translations of the LHS process `(e1->SKIP)`. Where the second TA is a translation of the prefix event `e1` using Rule 3.5. And the third TA is a translation of the subsequent process `SKIP` using Rule 3.3. Fourth and fifth TA are translations of the RHS process `(e2->SKIP)`. Fourth TA is a translation of the prefix event `e2` using Rule 3.5. While, the fifth TA is a translation of the subsequent process `SKIP` using Rule 3.3. Finally, the last TA is an environment TA for the list of the translated

TA. This completes the list of the output TA that capture the behaviour of the process `((e1->SKIP)|-|(e2->SKIP))`.

3.3.9. Translation of External Choice

This section describes the translation of the construct External Choice. The section begins with presenting a rule for translating the operator for external choice and then follows with an example that illustrates using the rule in translating a process.

Rule 3.9 defines a translation of external choice. The operator of external choice ([]) is another binary operator that combines two processes `P1` and `P2`. Rule 3.9 translates the operator external choice into a TA that coordinates a lists of translated TA: `Tp1` and `Tp2` for the translation of the two processes `P1` and `P2`, respectively.

In Figure 34, we annotate the structure of the output TA with the names used in the translation rule. The output TA has three transitions, and three locations defined in Lines 7–10 and 12–15, respectively. Then, Lines 17–21 define the corresponding labels of the transitions. Lines 27 extracts the initials of the external choice and then updates the initials in Lines 31–36. Finally, Lines 39–42 define the subsequent translation of processes `P1` and `P2`, which produces list of TA: `Tp1` and `Tp2`, respectively.

The behaviour of the output TA begins on transition `tran1` with performing a flow action (defined in Line 12). Then, on both transition `tran2` (Line 13) and `tran3` (Line 14) the TA performs two additional flow actions that activate two list of TA: `Tp1` and `Tp2` define in Lines 39–34 and 41–42 respectively. Thus, the output TA activates both `Tp1` and `Tp2` simultaneously, which makes the behaviour of both `Tp1` and `Tp2` available to the environment, such that choosing one of the translated list of TA blocks the other alternative list of TA. That is, choosing `Tp1` blocks `Tp2`, likewise choosing `Tp2` blocks `Tp1`.

An essential part of translating external choice is translating both processes such that choosing one process blocks the behaviour of the other process. We achieved this, with additional transitions in the first TA of each of the translated processes for the external choice, as discussed in Section 3.2. In the parameters of the translation function `transTA`, the parameter `usedNames` contains the initials of the processes for external choices, which is updated in Lines 31–36, and then use in translating each process, specifically in constructing the transition of blocking external choice that has co-actions (initials of the other processes) for blocking the process that is not chosen by the environment. The following Example 3.12 illustrates using this rule in translating a process that composes two processes with the operator of external choice.

Rule 3.9. Translation of External Choice

```
transTA (ExtChoice p1 p2) procName bid sid fid usedNames =
   ([(TA idTA [] [] locs [] (Init loc1) trans )] ++ ta1 ++ ta2,
   (sync1 ++ sync2), (syncMap1 ++ syncMap2) )
   where
      idTA  = "taIntCho" ++ bid ++ show sid

      loc1  = Location "id1" "s1" EmptyLabel None
      loc2  = Location "id2" "s2" EmptyLabel CommittedLoc
      loc3  = Location "id3" "s3" EmptyLabel CommittedLoc
      locs  = [loc1, loc2, loc3]

      tran1 = Transition loc1 loc2 [lab1]
      tran2 = Transition loc2 loc3 [lab2]
      tran3 = Transition loc3 loc1 [lab3]
      trans = [tran1, tran2, tran3]

      lab1  = Sync (VariableID (startEvent procName bid sid) [])
          Ques
      lab2  = Sync (VariableID
                     ("startID" ++ (bid ++ "0") ++ show (sid+1)
                        ) [])
                   Excl
      lab3  = Sync (VariableID
                     ("startID" ++ (bid ++ "1") ++ show (sid+2)
                        ) [])
                   Excl

      -- Extract a list of names for external choice from the
      -- parameter usedNames.
      (syncEv, syncPoint, hide, rename, exChs, intrr, iniIntrr,
              excps) = usedNames

     -- Updates the used names for subsequent translation
      exChs'  = exChs ++ (initials p2)
      exChs'' = exChs ++ (initials p1)
      usedNames'  = (syncEv, syncPoint, hide, rename, exChs',
         intrr,  iniIntrr, excps)
      usedNames'' = (syncEv, syncPoint, hide, rename, exChs'',
         intrr, iniIntrr, excps)

      -- translation of RHS and LHS processes p1 and p2
      (ta1, sync1, syncMap1) =
           transTA p1 [] (bid ++ "0") (sid+1) fid usedNames'
      (ta2, sync2, syncMap2) =
           transTA p2 [] (bid ++ "1") (sid+2) fid usedNames''
```

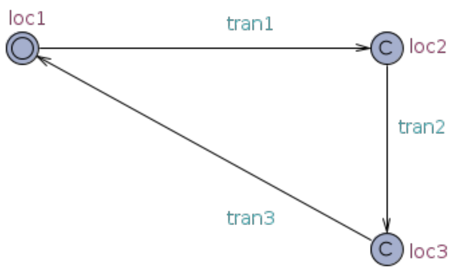
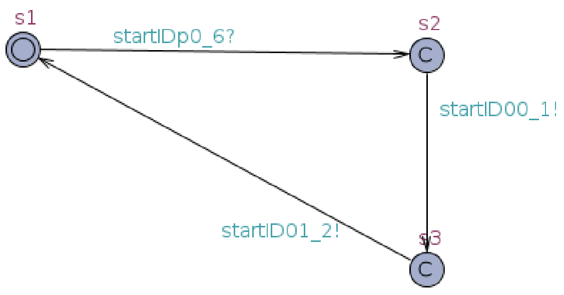


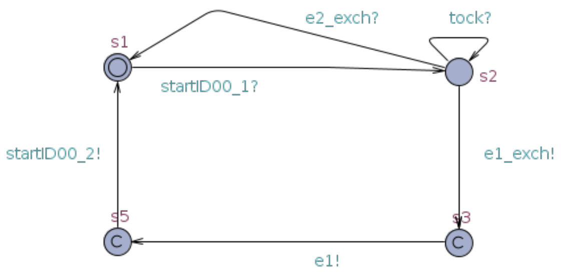
Figure 34: A structure of the control TA for the translation of external choice.

Example 3.12. An example of translating a process that composes two processes with the operator of external choice.

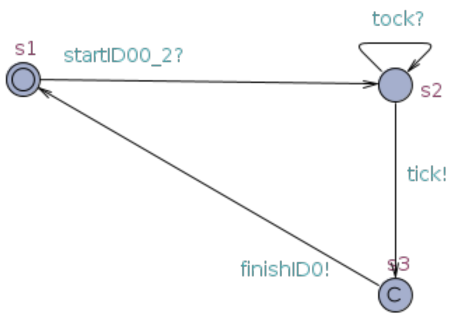
```
transTA((e1->SKIP)[](e2->SKIP))
  = [
```



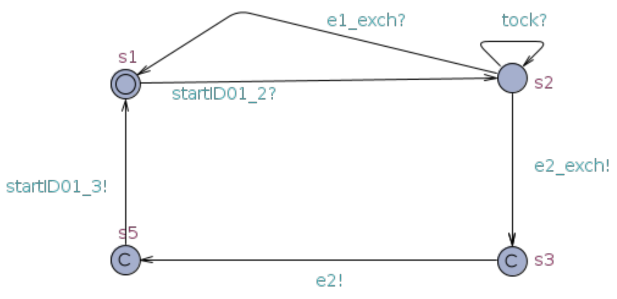
```
        ] ++ ta1 ++ ta2
where
  ta1 = transTA(e1->SKIP)
      = [
```



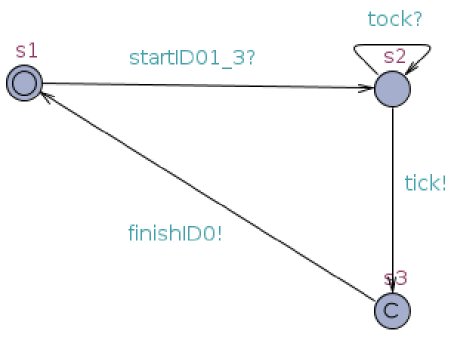
```
        ] ++ transTA(SKIP)
        = [
```

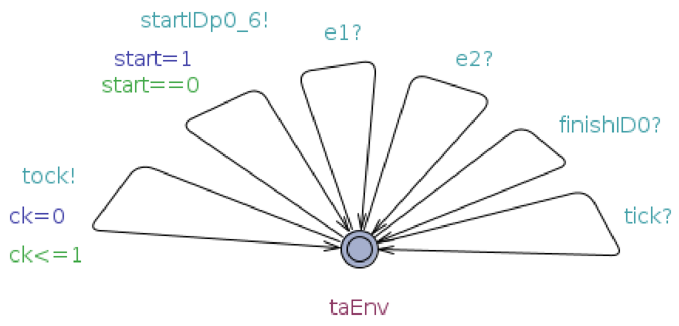


```
               ]
       ta2 = transTA(e2->SKIP)
           = [
```



```
               ] ++ transTA(SKIP)
                  = [
```





]

Example 3.12 illustrates using Rule 3.9 in translating a process `((e1->SKIP)[](e2->SKIP))` into a list of TA that contains 5 TA. The first TA is a translation of the operator external choice. The TA has three transitions, each labelled with a flow action, `startIDp0_6` `startID00_1` and `startId01_2`. Initially, the behaviour of the TA synchronises on the first flow action `startIDp0_6` and then immediately performs the two subsequent flow actions `startID00_1` and `startId01_2` that activates the list of TA for the translations of the processes: `(e1->SKIP)` and `(e2->SKIP)`, respectively.

Second and third TA are translations of the LHS process `e1->SKIP`. Second TA is a translation the event `e1`, which synchronises on the flow action `startID00_1` and moves to location `s2` where the TA has three possible transitions: `e1_exch?` `e2_exch?` and `tock?`. On the transition `tock?`, the TA performs the action `tock` for the progress of time. On transition `e2_exch?` the TA performs a blocking event when the environment chooses the other action `e2`. Lastly, on transition `e1_exch!` the TA performs the action `e1_exch!` when the environment chooses the action `e1` for the behaviour `Tp1`. First, the action `e1_exch!` synchronise with its co-action `e1_exch?` to block the alternative behaviour of `Tp2`, and then immediately proceeds with performing the chosen action `e1` that leads to the subsequent flow action `startID00_2`, which activates the subsequent TA third TA. The third TA is a translation of the subsequent process `SKIP` for the LHS process `e1->SKIP` translated with Rule 3.3.

Fourth and Fifth TA are translations of the RHS process `(e2->SKIP)`. Fourth TA is a translation of the event `e2` using Rule 3.5, similar to the previous translation of the first TA; and the fifth TA is a translation of the remaining process `SKIP` using Rule 3.3. Finally, the last TA is an environment TA for the list of the translated TA, as defined in Section 3.2. This completes the description of translating the process `(e1->SKIP)[](e2->SKIP)` into a list of TA.

3.3.10. Translation of Sequential Composition

This section describes a translation of the operator sequential composition. The section begins with presenting a rule for translating the operator sequential composition; and then follows with an example that illustrates using the rule in translating a process.

Rule 3.10. Translation of Sequential Composition

```
transTA (Seq p1 p2) procName bid sid fid usedNames =
    ([(TA idTA [] [] locs [] (Init loc1) trans )] ++ ta1 ++ ta2,
    (sync1 ++ sync2), (syncMap1 ++ syncMap2) )
    where
      idTA = "taSequen" ++ bid ++ show sid

      loc1  = Location "id1" "s1" EmptyLabel None
      loc2  = Location "id2" "s2" EmptyLabel CommittedLoc
      loc3  = Location "id3" "s3" EmptyLabel None
      loc4  = Location "id4" "s4" EmptyLabel CommittedLoc
      locs  = [loc1, loc2, loc3, loc4]

      tran1 = Transition loc1 loc2 [lab1] []
      tran2 = Transition loc2 loc3 [lab2] []
      tran3 = Transition loc3 loc4 [lab3] []
      tran4 = Transition loc4 loc1 [lab4] []
      trans = [tran1, tran2, tran3, tran4]

      lab1  = Sync (VariableID
                      (startEvent procName bid sid) [])
                   Ques
      lab2  = Sync (VariableID
                       ("startID" ++ (bid ++ "0") ++ show (sid
                          +1)) [])
                   Excl
      lab3  = Sync (VariableID ("finishID" ++ show (fid+1)) [])
                   Ques
      lab4  = Sync (VariableID
                       ("startID" ++ (bid ++ "1") ++ show (sid
                          +2)) [])
                   Excl

        -- translation of the LHS process
      (ta1, sync1, syncMap1) =
            transTA p1 [] (bid ++ "0") (sid+1) (fid+1) usedNames

       -- translation of the RHS process
      (ta2, sync2, syncMap2) =
             transTA p2 [] (bid ++ "1") (sid+2) fid usedNames
```

This operator for sequential composition is another binary operator that composes two processes P1 and P2 sequentially. Like the previous translation rules, this rule translates the operator sequential composition into a control TA, which coordinates the list of TA Tp1 and Tp2 for the translation of the two processes P1 and P2.

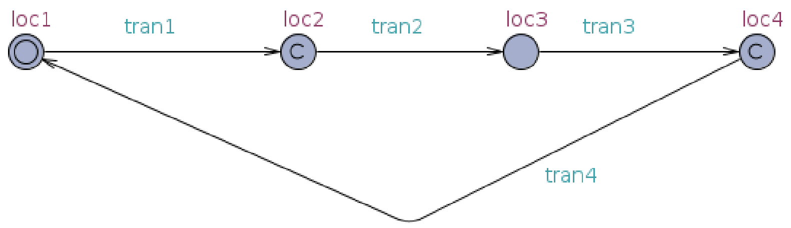
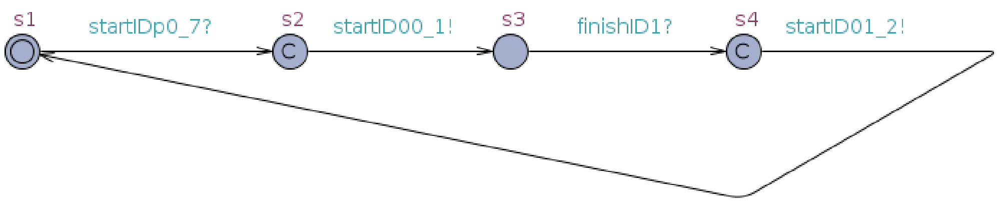


Figure 35: A structure of the control TA for the translation of sequential composition.

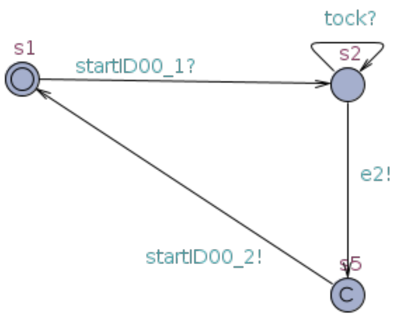
In Figure 35, we annotate the structure of the output TA with the names used in the translation Rule 3.10. The TA has four locations and four transitions that are defined in Lines 7–11 and Lines 13–17, respectively. In Rule 3.10, the behaviour of the output TA begins with synchronising on the first flow action (Line 13) and then immediately performs another two flow action on transition `tran2` (Line 14) to activate the translation of the LHS process `Tp1`. After that, the control TA waits on location `loc3` until the TA synchronises on a terminating action on transition `tran3` (Line 15), which indicates the termination of the first process `Tp1` and then immediately activates `Tp2` which proceeds up to its termination point. The following Example 3.13 illustrates using this rule in translating a process.

Example 3.13. An example for translating a process that composes two processes with the operator for sequential composition.

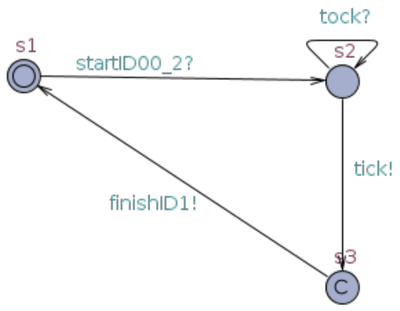
```
transTA((e2->SKIP);(e1->SKIP)) = [
```



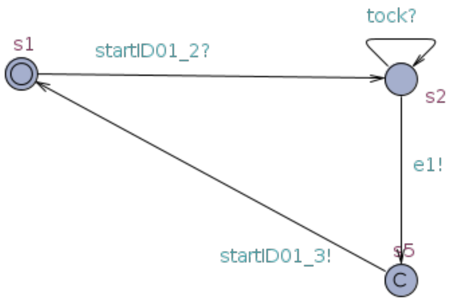
```
     ] ++ ta1 ++ ta2
where
ta1 = transTA(e2->SKIP)
     = [
```



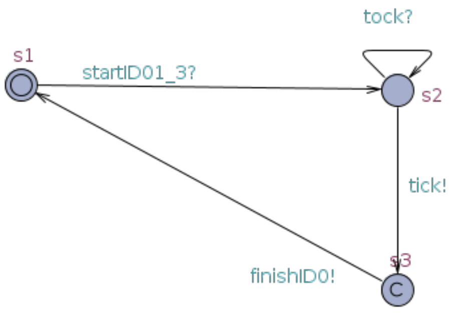
```
  ] ++ transTA(SKIP)
    = [
```

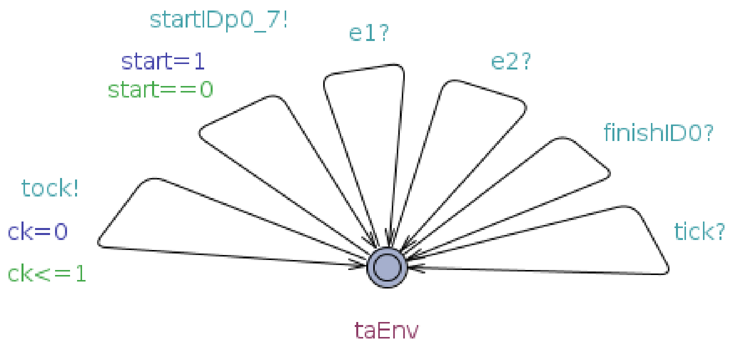


```
         ]
ta2 = transTA(e1->SKIP)
    = [
```



```
    ] ++ = transTA(SKIP)
         = [
```





13]

Example 3.13 illustrates using Rule 3.10 in translating the process `((e2->SKIP);(e1->SKIP))` into a list of TA that contains 5 TA. The first TA is a translation of the operator sequential composition using Rule 3.10. The second and third TA are translations of the LHS process `(e2-> SKIP)`, while the fourth and fifth TA are translations of the RHS process `(e1-> SKIP)`. Finally, the last TA is an environment TA for the list of the translated TA.

Details behaviour of the list of the translated TA is as follows. The first TA synchronises on the flow action `startIDp0_7?` and then immediately performs another flow action `startID00_1` to activate the second TA, and then waits on location `s3` until the TA synchronises on the termination action `finishID1?`; which indicates the termination of the LHS process `(e2->SKIP)`, and then immediately the TA performs the subsequent flow action `startID01_2` that activates the fourth TA for the translation of the RHS process. In the like manner, the fourth TA synchronises on the flow action and then performs the action `e1`, which follows with the subsequent flow action `startID01_3` that activates the fifth TA, which synchronises on the flow action and then performs the action `tick`, and then follows with a termination action `finishID0`, which synchronises with the co-action in the environment TA to indicate successful termination of the whole process. This completes the description of translating the process `((e2->SKIP);(e1->SKIP))` into a list of TA.

3.3.11. Translation of Generalised Parallel

This section describes a translation of the operator generalised parallel. The section begins with a rule for translating the operator generalised parallel; and then follows with an example that illustrates using the rule in translating a process.

Rule 3.11. Translation of Generalised Parallel

```
transTA (GenPar p1 p2 es) procName bid sid fid usedNames =
 ([(TA idTA [] [] locs [] (Init loc1) trans )] ++ ta1 ++ ta2,
 (es ++ sync1 ++ sync2), (syncMap1 ++ syncMap2) )
 where
  idTA   = "taGenPar" ++ bid ++ show sid
  loc1   = Location  "id1" "s1" EmptyLabel None
  loc2   = Location  "id2" "s2" EmptyLabel CommittedLoc
  loc3   = Location  "id3" "s3" EmptyLabel CommittedLoc
  loc4   = Location  "id4" "s4" EmptyLabel CommittedLoc
  loc5   = Location  "id5" "s5" EmptyLabel None
  loc6   = Location  "id6" "s6" EmptyLabel None
  loc7   = Location  "id7" "s7" EmptyLabel None
  loc8   = Location  "id8" "s8" EmptyLabel CommittedLoc
  locs   = [loc1, loc2, loc3, loc4, loc5, loc6, loc7, loc8]
  tran1  = Transition loc1 loc2 [lab1] []
  tran2  = Transition loc2 loc3 [lab3] []
  tran3  = Transition loc3 loc5 [lab2] []
  tran4  = Transition loc2 loc4 [lab2] []
  tran5  = Transition loc4 loc5 [lab3] []
  tran6  = Transition loc5 loc6 [lab4] []
  tran7  = Transition loc5 loc7 [lab5] []
  tran8  = Transition loc6 loc8 [lab5] []
  tran9  = Transition loc7 loc8 [lab4] []
  tran10 = Transition loc8 loc1 [lab6] []
  trans  = [tran1, tran2, tran3, tran4, tran5,
                 tran6, tran7, tran8, tran9, tran10]
  lab1 = Sync (VariableID (startEvent procName bid sid) []) Ques
  lab2 = Sync (VariableID ("startID" ++ (bid ++ "0") ++
                                show (sid+1)) []) Excl
  lab3 = Sync (VariableID ("startID" ++ (bid ++ "1") ++
                                show (sid+2)) []) Excl
  lab4 = Sync (VariableID ("finishID" ++ show (fid+1 )) []) Ques
  lab5 = Sync (VariableID ("finishID" ++ show (fid+2 )) []) Ques
  lab6 = Sync (VariableID ("finishID" ++ show  fid    ) []) Excl

  (syncEv, syncPoint, hide, rename, exChs, intrr, iniIntrr,
                  excps) = usedNames
  syncEv'    = es ++ syncEv             -- Update synch name
  usedNames' = (syncEv', syncPoint, hide, rename, exChs, intrr,
     iniIntrr, excps)
  (ta1, sync1, syncMap1) =
  transTA p1 [] (bid ++ "0") (sid+1) (fid+1) usedNames'
      (ta2, sync2, syncMap2) =
  transTA p2 [] (bid ++ "1") (sid+2) (fid+2) usedNames'
```

The operator generalised parallel is another binary operator, which is composed of two processes P1 and P2 that run in parallel and synchronise on a specified set of synchronisation events. The construct GenPar is translated into two TA: a control TA and a synchronisation TA. The synchronisation TA (Definition 3.10) coordinates the synchronisation of the translated processes Tp1 and Tp2. While the control TA coordinates the behaviour of the translated processes Tp1 and Tp2.

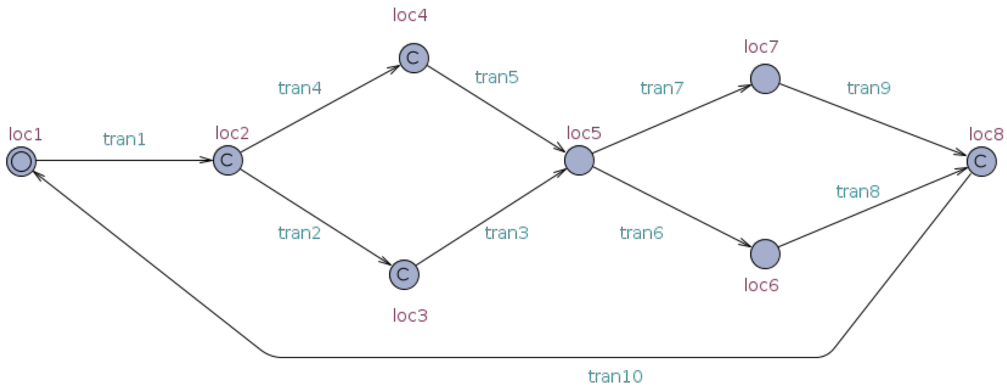


Figure 36: A structure of TA for the translation of operator Generalised Parallel

In Figure 36, we annotate the structure of the control TA with the names used in the translation rule. The structure of the control TA has 8 locations and ten transitions, as defined in Lines 6–14 and Lines 15–26 respectively. Lines 27–34 define the labels of the transitions. Lines 36–37 extracts the used names from the parameter usedNames for another new name that contains an updated list of synchronisation names syncEv in Lines 38–40. Then, Lines 41–44 is a recursive call for the translation of the processes P1 and P2, which produces the list of TA, Tp1 and Tp2.

The behaviour of the control TA begins on tran1 for synchronising on a flow action and then immediately performs another two flow actions to activate both Tp1 and Tp2 simultaneously, that is on tran2 and tran3 or tran3 and tran4 in both two possible orders depending on the environment, that is either Tp1 simultaneously with Tp2 or Tp2 simultaneously with Tp1. Then, the control TA waits on location s4 until either Tp1 or Tp2 terminates and then waits for the other TA to terminate, depending on the first process that terminates, either Tp1 and then Tp2 or Tp2 and then Tp1. After that, the control TA immediately performs another termination action, which records the termination of the whole process.

The translated processes synchronise on a multi-synchronisation action, which synchronises more than two translated TA: at least two translated processes and the Environment TA. As highlighted in the translation strategy (Section 3.2), in handling multi-synchronisation, we adopt a centralised approach [114, 115] for using a separate controller in handling multi-synchronisation. In this work, we implement the approach in a functional style with Haskell. In Definition 3.10, the function syncTA coordinates the synchronisation of multi-synchronisation actions.

In translating the multi-synchronisation, each translated process that participates in multi-synchronisation has a client TA, which synchronises on a multi-synchronisation action. The client TA sends a synchronisation request to the synchronisation controller TA `syncTA` and then waits for a synchronisation response.

When the synchronisation controller receives all the synchronisation requests, which indicates that all the synchronisation clients are ready for the synchronisation, then a guard for performing the multi-synchronisation action is enabled, and `syncTA` communicates the multi-synchronisation action to the environment and then also immediately broadcast the multi-synchronisation action that responds to all the awaiting client TA.

On receiving the broadcast multi-synchronisation response, all the awaiting client TA synchronise and proceed. An example of using this rule in translating a process is illustrated in the following Example 3.14, which demonstrates using the rule in translating concurrent processes that are composed with the operator generalised parallel `GenPar` in the specification of the process.

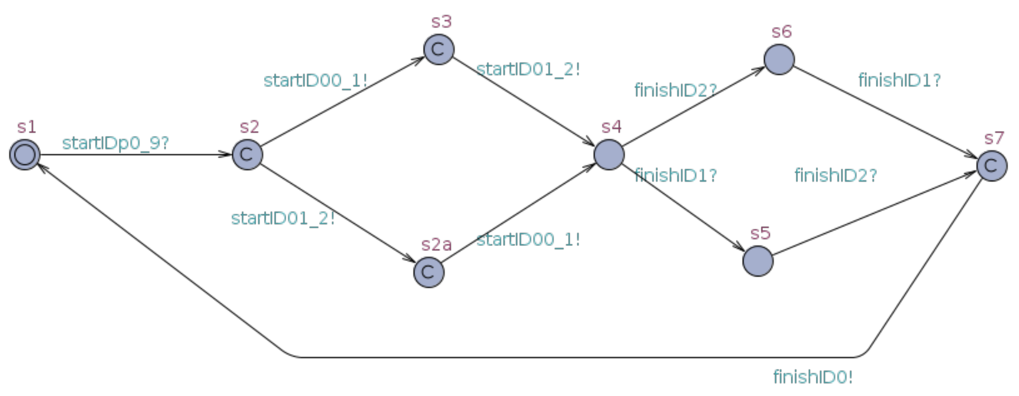
In Definition 3.10, we define a function for the synchronisation TA `syncTA` which takes two parameters, a list of synchronisation actions and a list of pairs that assign an identifier to each synchronisation action. The output of the function is a TA with an identifier "syncTA" (Line 3). The output TA has one starting location and one location for each synchronisation action as defined in Definition 3.10 (Lines 4–7). Line 8 defines a function for generating the transitions of the TA. Each synchronisation action has two transitions one from the initial location and the second transition back to the initial location. The first transition has a guard that is only enabled when all the synchronisation TA becomes ready for the synchronisation. This is illustrated in Example 3.14.

Definition 3.10. Synchronisation TA

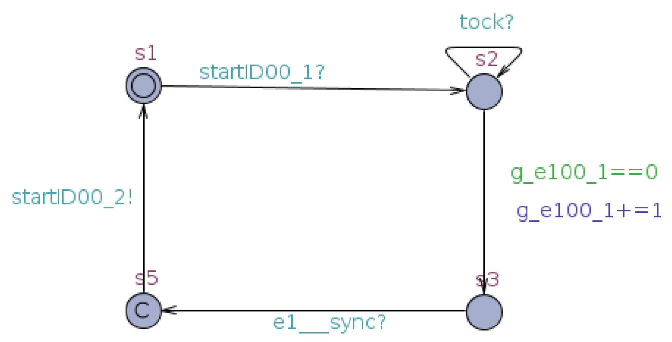
```
syncTA :: [Event] -> [SyncPoint] -> TA
syncTA     events     syncMaps   =
    TA "SyncTA" [] [] (loc:locs) [] (Init loc) trans
    where
      loc  = Location "SyncPoint" "SyncPoint" EmptyLabel None
      locs = [(Location ("s"++ show e) ("s"++ show e) EmptyLabel
                         CommittedLoc) | e <- uniq events]
      trans = transGen  loc (uniq events) syncMaps events

-- Generates transitions for the sync controller
    transGen :: Location->[Event]->[SyncPoint]->[Event]->[
       Transition]
    transGen    l0        []        _            _     = []
    transGen    l0        (e:es)   syncMaps     syncs  =
        [(Transition
           l0 l
           (Sync (VariableID (show e) []) Excl),
           (Guard
              (ExpID
                ((addExpr
                  [("g_" ++ tag)|(e1, tag) <- syncMaps, e == e1
                     ])
                  ++ "␣==␣" ++
                  show ((length [e1 | e1 <- syncs, e == e1 ]) +
                     1) ) ) ),
           (Update ([ AssgExp (ExpID ("g_" ++ tag))
                       ASSIGNMENT (Val 0) |(e1, tag) <- syncMaps,
                       e == e1] )) ] [])]
        ++ [Transition
             l l0
             [(Sync (VariableID ((show e) ++ "___sync") []) Excl
                )] [] ]
        ++ (transGen l0 es syncMaps syncs)
        where
          l = (Location ("s"++ show e) ("s"++ show e)
                         EmptyLabel CommittedLoc)
```

Example 3.14. An example that illustrates using both Rule 3.11 and the definition of synchronisation controller (Definition 3.10) in translating a process.

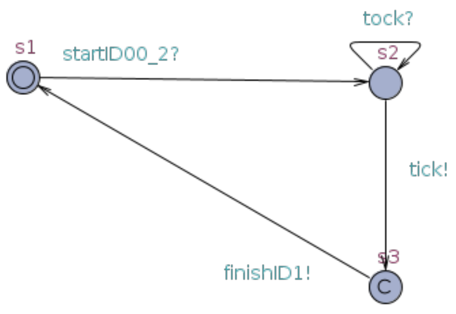
```
transTA((e1->SKIP)[|{e1}|](e1->SKIP)) = [
```



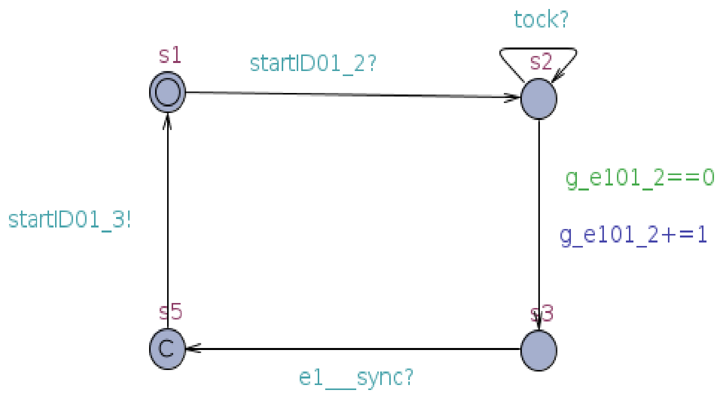
```
        ] ++ ta1 ++ ta2

where
ta1 = transTA(e1->SKIP)
     = [
```



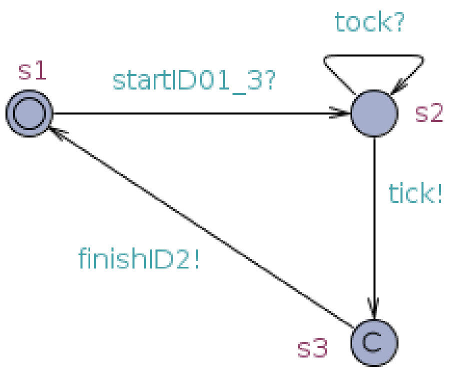
```
  ] ++ transTA(SKIP)
      = [
```

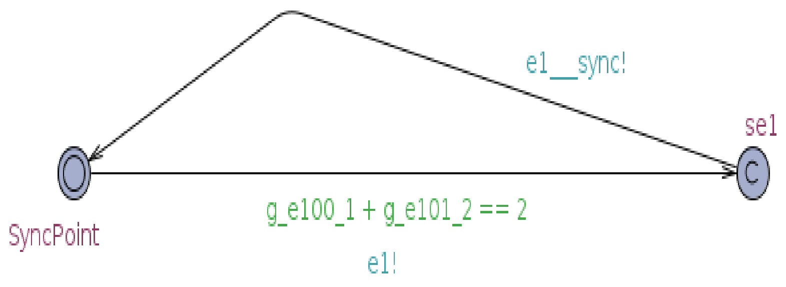


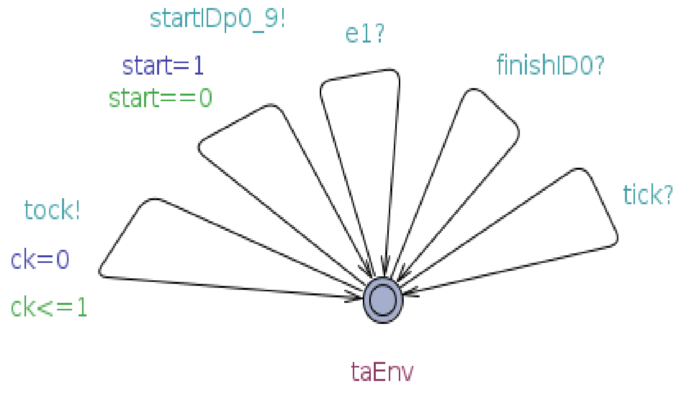
```
         ]

ta2 = transTA(e2->SKIP)
    = [
```



```
      ] ++ transTA(SKIP)
        = [
```







]

Example 3.14 illustrates using Rule 3.11 in translating the process `(e1->SKIP)[|{e1}|](e1->SKIP)` into a list of TAs that contains seven TAs. The first TA captures the translation of the operator generalised parallel. The second and third TA captures the translation of the LHS process. The fourth and fifth TA captures the translation of the RHS process. The sixth TA is a synchronisation TA that coordinates the synchronisation of the action `e1`. Finally, the last TA is an environment TA for the list of the translated TA for the process `(e1->SKIP)[|{e1}|](e1->SKIP)`.

The behaviour of the translated TA is as follows. The first TA is the control TA that initially synchronises on the first flow action `startIDp0_9` and then immediately performs two flow actions in two possible orders, either `startID00_1` and then `startID01_2` or `startID01_2` and then `startID00_1`, depending on the environment. Then the control TA waits on location `s4` until the control TA synchronises on the termination action `finishID2` and then synchronise on the second termination action `finishID1`, for the LHS and RHS processes respectively. Alternatively, the control TA synchronises first on `finishID1` and then synchronises on the termination action `finishID2`, depending on the process that terminates first, either the LHS process or the RHS process.

The second TA synchronises on the flow action `startID00_1` and then updates its guard to indicate its readiness to synchronise on the multi-synchronisation action `e1`. Then, on receiving a response for the synchronisation, the TA synchronises on the broadcast multi-synchronisation action `e1____sync?`, which enables the TA to proceed with performing another flow action that activated the third TA, which captures the translation of the subsequent process `SKIP` as described in Rule 3.3. Similarly, the fourth and fifth TA captures the translation of the RHS process `(e1->SKIP)`.

3.3.12. Translation of Interleaving

This section describes the translation of the operator interleaving. The section begins with presenting a rule for translating the operator interleaving and then follows with an example that illustrates using the rule in translating a process.

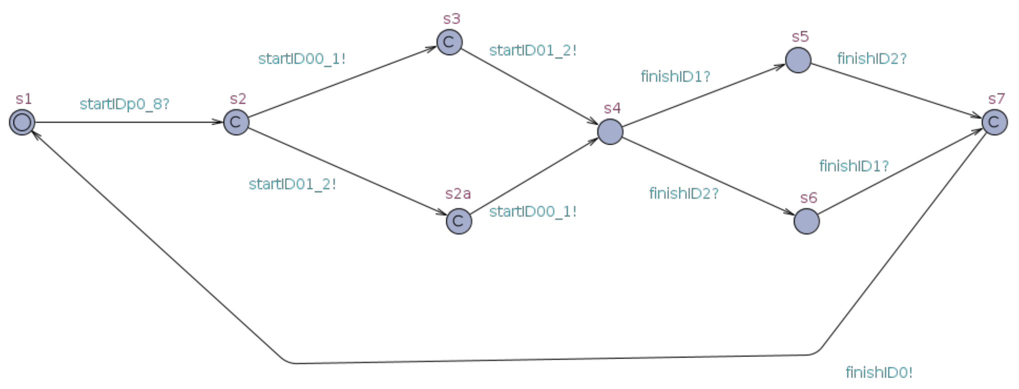
Rule 3.12. Translation of Interleaving

```
1 transTA (Interleave p1 p2)      procName bid sid fid usedNames
2   = transTA (GenPar  p1 p2 []) procName bid sid fid usedNames
3               -- As generalised parallel with empty synch events
```

The operator interleaving is translated in terms of the constructor for generalised parallel with empty synchronisation events. In *tock-CSP*, this is expressed as `(P1 ||| P2) = (P1 |[{}]| P2)`. Line 1 defines the function `transTA` for the construct `Interleave` and the required parameters. While line 2 defines the output TA in terms of the construct for the generalised parallel `GenPar` with empty synchronisation events. The following example illustrates using the rule in translating a process.

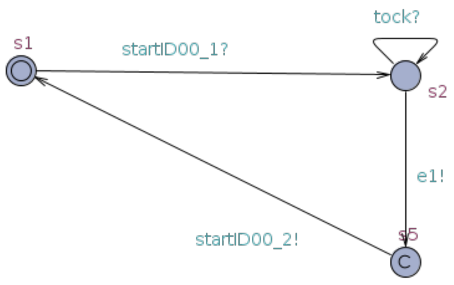
Example 3.15. An example of translating a *tock-CSP* process that composes processes with the operator interleaving using Rule 3.12.

```
  transTA((e1->SKIP)|||(e1->SKIP)) = [
```

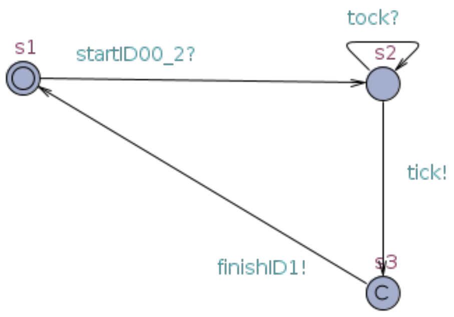


```
      ] ++ ta1 ++ ta2
where
```

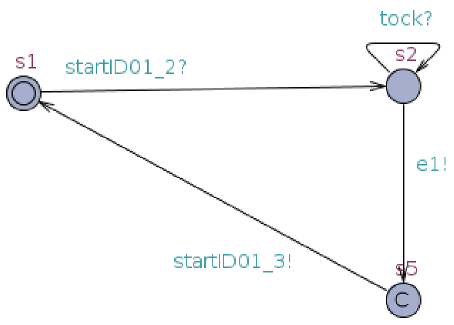
```
ta1 = transTA(e1->SKIP)
    = [
```



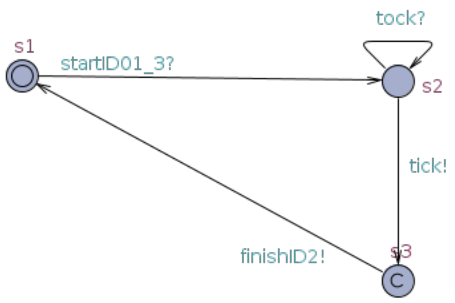
```
  ] ++ transTA(SKIP)
     = [
```

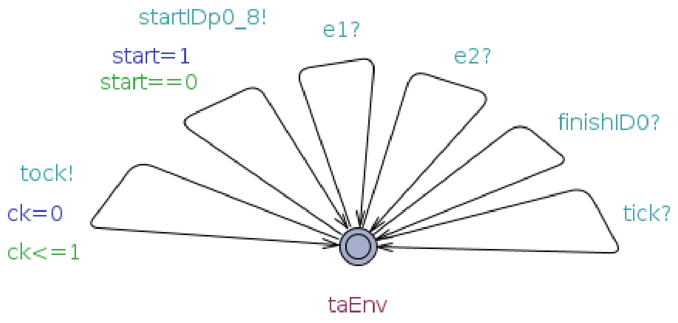


```
             ]
ta2 = transTA(e1->SKIP)
    = [
```



```
        ] ++ transTA(SKIP)
           = [
```





]

Example 3.15 illustrates using Rule 3.12 in translating a process into a list containing 6 TA. The first TA is a translation of the operator Interleaving. Second and third TA are translations of the LHS process `(e1->SKIP)`. Fourth and fifth TA are translations of the RHS process `e1->SKIP)`. The last TA is an environment TA for the list of the translated TA.

The behaviour of the TA is as follows. The first TA synchronises on the coordinating start event `startIDp0_8` and then immediately performs two flow actions `starID00_1` and `startID01_2` simultaneously that activate the translation of the LHS and RHS processes respectively. And then the first TA waits on location `s3` until it synchronises on a termination action, either `finishID1` or `finishID2`, and then waits for the second termination action `finishID1` or `finishID2` depending on the first terminating process. The action `finishID1` is termination action of the translated LHS process `(e1->SKIP)`, and `finishID1` is the termination action of the translated RHS process `(e1->SKIP)`. Then, the first TA performs another termination action to record the termination of the whole process.

The second TA is a translation of the event `e1` using Rule 3.5. While, the third TA is a translation of the subsequent process `SKIP` using Rule 3.3. Also, TA3 is a translation of the event `e2` using Rule 3.5. While TA4 is a translation of the subsequent process `SKIP` using Rule 3.3. This completes the description the translated TA in Example 3.15 that demonstrates using Rule 3.12 in translating the process `(e1->SKIP)|||(e1->SKIP)` into a list of TAs.

3.3.13. Translation of Interrupt

This section describes the translation of the operator Interrupt. The section begins with presenting a rule for translating the operator `Interrupt` and then follows with an example that illustrates using the rule in translating a process.

Rule 3.13. Translation of Interrupt

```
transTA (Interrupt p1 p2 ) procName bid sid fid usedNames =
    ([(TA idTA [] [] locs [] (Init loc1) trans )] ++ ta1 ++ ta2,
        (sync1 ++ sync2), (syncMap1 ++ syncMap2) )
    where
       idTA  = "taIntrpt" ++ bid ++ show sid

       loc1  = Location "id1" "s1" EmptyLabel None
       loc2  = Location "id2" "s2" EmptyLabel CommittedLoc
       loc3  = Location "id3" "s3" EmptyLabel CommittedLoc
       locs  = [loc1, loc2, loc3]

       tran1 = Transition loc1 loc2 [lab1] []
       tran2 = Transition loc2 loc3 [lab2] []
       tran3 = Transition loc3 loc1 [lab3] []
       trans = [tran1, tran2, tran3]

       lab1  = Sync (VariableID (startEvent procName bid sid)
           [])   Ques
       lab2  = Sync (VariableID ("startID" ++ (bid ++ "0") ++
              show (sid+1)) []) Excl
       lab3  = Sync (VariableID ("startID" ++ (bid ++ "1") ++
              show (sid+2)) []) Excl

       (syncEv, syncPoint, hide, rename, exChs, intrr, iniIntrr,
                excps) = usedNames

       -- Updates the parameters for interrupts
       intrr'    = intrr ++ (initials p2)
       iniIntrr' = iniIntrr ++ (initials p2)
       usedNames'  = (syncEv, syncPoint, hide, rename, exChs,
                      intrr', iniIntrr, excps)
       usedNames'' = (syncEv, syncPoint, hide, rename, exChs,
                      intrr, iniIntrr', excps)

       (ta1, sync1, syncMap1) =
             transTA p1 [] (bid ++ "0") (sid+1) fid usedNames'
       (ta2, sync2, syncMap2) =
             transTA p2 [] (bid ++ "1") (sid+2) fid usedNames''
```

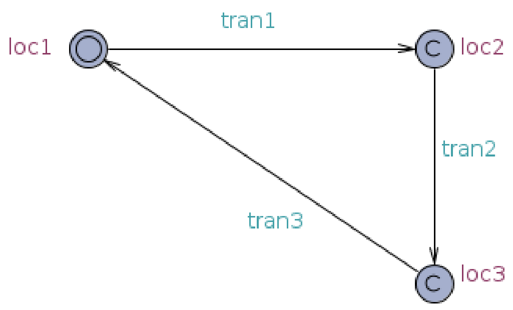


Figure 37: A TA for the structure of the translation of the operator interrupt.

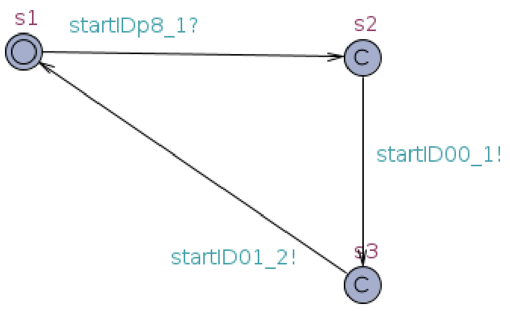
The operator Interrupt is also another binary operator that comprises two processes `P1` and `P2` in such a way that the process `P1` begins its behaviour that can be interrupted by process `P2`, whenever process `P2` performs an event. The operator Interrupt is translated into a TA that coordinates the list of TA for the translated processes `P1` and `P2` into `Tp1` and `Tp2`, respectively.

In Figure 37, we annotate the structure of the translated TA with the names used in the translation rule. The translated TA has 3 locations and three transitions defined in Lines 7–10 and Lines 12–15. Lines 17–22 define labels for the transitions. Line 24 extracts the list of used names for interrupt. Lines 28–29 provides a new name for the updated list of the initials of the interrupting process `p2`. Also, Lines 31–34 provides a new name for the updated tuples of the used names `usedNames`. Lines 36–40 define the subsequent translation of the LHS and RHS processes, that is `Tp1` and `Tp2`, respectively.

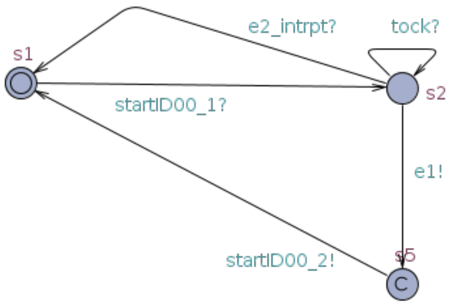
The behaviour of the control TA begins on transition `tran1` for performing a flow action. And then immediately activates the translation of the processes `Tp1` and `Tp2`. For the translation of the interrupted process `Tp1`, each TA in the list `Tp1` has an additional interrupting transition in each stable location, as described in the Translation strategy. The additional transition provides a co-action of the initials of the interrupting process `Tp2`, which enables `Tp2` to interrupt `Tp1` at any stable location. The following example 3.16 demonstrates using the rule in translating a process.

Example 3.16. An example of using Rule 3.13 in translating the process `((e1-> SKIP)/\(e2-> SKIP))` into a list of TA as follows.

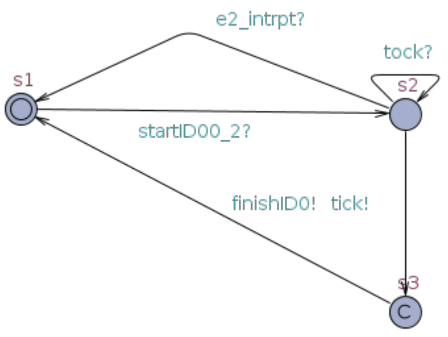
```
transTA((e1-> SKIP)/\(e2-> SKIP)) = [
```



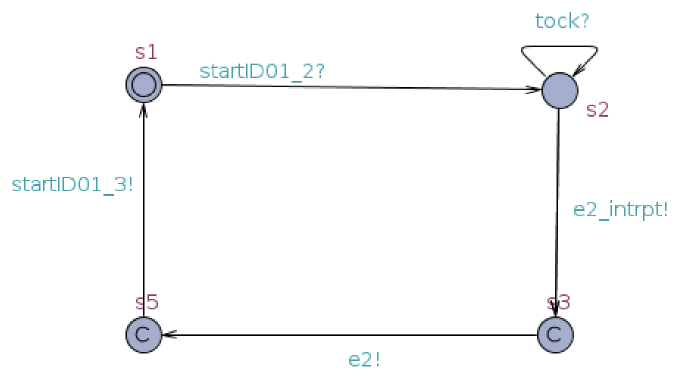
```
        ] ++ ta1 ++ ta2

where
ta1 = transTA(e1->SKIP)
     = [
```



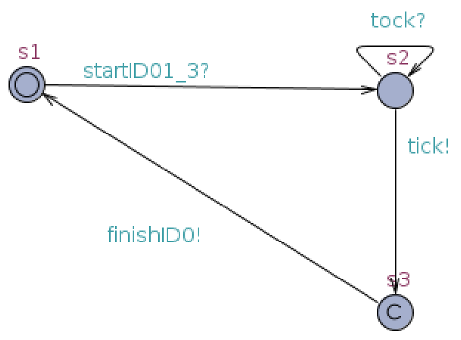
```
   ] ++ transTA(SKIP)
      = [
```

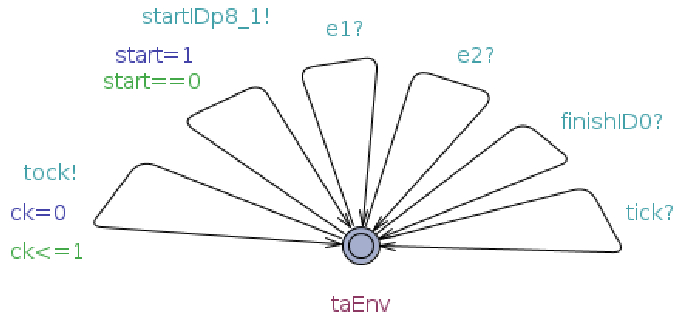


```
                ]

ta2 = transTA(e2->SKIP)
      = [
```



```
         ] = transTA(SKIP)
           = [
```





]

Example 3.16 demonstrates using Rule 3.13 a translation of the process into a list containing 5 TA. The first TA is a translation of the operator Interrupt. Second and third TA are translations of the LHS process (e1->SKIP). The second TA is a translation of the event e1 using Rule 3.5. And the third TA2 is a translation of the subsequent process SKIP using Rule 3.6. While the fourth and fifth TAs are the translation of the RHS process (e2->SKIP). Fourth TA is a translation of the event e2. Fifth TA is a translation of the subsequent process SKIP. The last TA is an environment TA for the list of translated TA.

The behaviour of the translated TA begins with the first TA that synchronises on the flow action startIDp8_1, and then immediately performs two subsequent flow actions startID00_1 and startID01_2 that activate the second and third TA. The second TA synchronises on the action startID00_1, then on location s2, the second TA can be interrupted with co-action of e2_intrpt, or perform the action tock to record the progress of time or proceeds to perform the action e1! and then immediately performs another flow action startID00_2 to activate the third TA.

The third TA synchronises on the flow action startID00_2, and then on location s2, also, the third TA can be interrupted with the co-action e2_intrpt, or perform the action tock to record the progress of time or progress to perform the action tick and then immediately perform the termination action finishID0 to record a successful termination of the LHS process without interrupt.

The fourth TA initiates the behaviour of the RHS process that interrupts the behaviour of the LHS process. The fourth TA begins with synchronising on a flow action stardID01_2 initiated by the first TA. Then, on location s2, either the TA performs the action tock or performs an interrupt action e2_intrpt to interrupt the behaviour of the LHS process, then proceeds to perform the action e2, and then performs another flow action startID01_3 to activate the fifth TA, which performs the action tick, and then performs the termination action finishID0, which records a successful termination of the process. This completes the description of the list of TAs that capture the behaviour of the process ((e1->SKIP)/\(e2->SKIP)).

3.3.14. Translation of Exception

This section discussed the translation of the operator Exception. The section begins with presenting a rule for translating the operator Exception and then follows with an example that illustrates using the rule in translating a process.

Rule 3.14. Translation of Exception

```
transTA (Exception p1 p2 es) procName bid sid fid usedNames =
 ([(TA idTA [] [] locs [] (Init loc1) trans )] ++ ta1 ++ ta2,
    (sync1 ++ sync2), (syncMap1 ++ syncMap2) )
 where
  idTA  = "taException" ++ bid ++ show sid
  loc1  = Location "id1" "s1" EmptyLabel None
  loc2  = Location "id2" "s2" EmptyLabel CommittedLoc
  loc3  = Location "id3" "s3" EmptyLabel None
  loc4  = Location "id4" "s4" EmptyLabel CommittedLoc
  loc6  = Location "id6" "s6" EmptyLabel CommittedLoc
  loc7  = Location "id7" "s7" EmptyLabel None
  loc8  = Location "id8" "s8" EmptyLabel CommittedLoc
  locs  = [loc1, loc2, loc3, loc4, loc6, loc7, loc8]
  tran1 = Transition loc1 loc2 [lab1] []
  tran2 = Transition loc2 loc3 [lab2] []
  tran3 = Transition loc3 loc4 [lab3] []
  tran4 = Transition loc4 loc1 [lab4] []
  tran5 = Transition loc3 loc6 [lab5] []
  tran6 = Transition loc6 loc7 [lab6] []
  tran7 = Transition loc7 loc8 [lab7] []
  tran8 = Transition loc8 loc1 [lab4] []
  trans = [tran1, tran2, tran3, tran4, tran5, tran6, tran7,
     tran8]
  lab1 = Sync (VariableID (startEvent procName bid sid) []) Ques
  lab2 = Sync (VariableID ("startID" ++ (bid ++ "0") ++
             show (sid+1)) []) Excl
  lab3 = Sync (VariableID ("finishID"  ++ show (fid+1)) []) Ques
  lab4 = Sync (VariableID ("finishID"  ++ show (fid))   []) Excl
  lab5 = Sync (VariableID ("startExcp" ++ show (fid+1)) []) Ques
  lab6 = Sync (VariableID ("startID"   ++ (bid ++ "1") ++
             show (sid+2)) []) Excl
  lab7 = Sync (VariableID ("finishID"  ++ show (fid+1)) []) Ques

  -- Assigns a new name for the updates list usednames
  (syncEv, syncPoint, hide, rename, exChs, intrr, iniIntrr,
     excps) = usedNames
  excps' = (((fst excps) ++ es), snd excps)
  usedNames' = (syncEv, syncPoint, hide, rename, exChs, intrr,
     iniIntrr, excps')

  -- Subsequent translation of the remaining processes
  (ta1, sync1, syncMap1) =
      transTA p1 [] (bid ++ "0") (sid+1) (fid+1) usedNames'
  (ta2, sync2, syncMap2) =
     transTA p2 [] (bid ++ "1") (sid+2) (fid+1) usedNames'
```

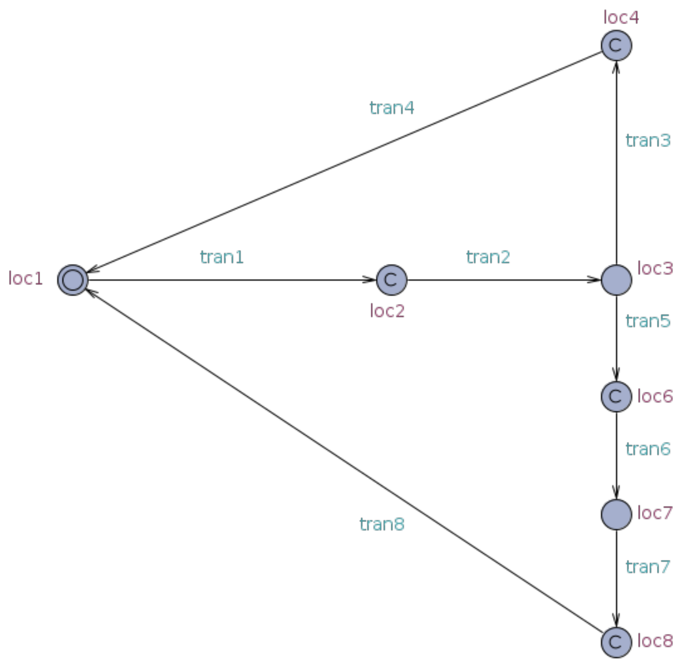


Figure 38: A structure of the control TA for the translation of the operator Exception.

The operator Exception is also another binary operator that combines two processes `p1` and `p2`. Initially the process behaves as `p1` until either `p1` terminates or performs an exception event from the list `es` which terminates the process `p1` and initiates the process `p2`. Like the previous binary operators, the operator `Exception` is translated into a control TA that coordinates the translation of the two processes `p1` and `p2` into `Tp1` and `Tp2`, respectively.

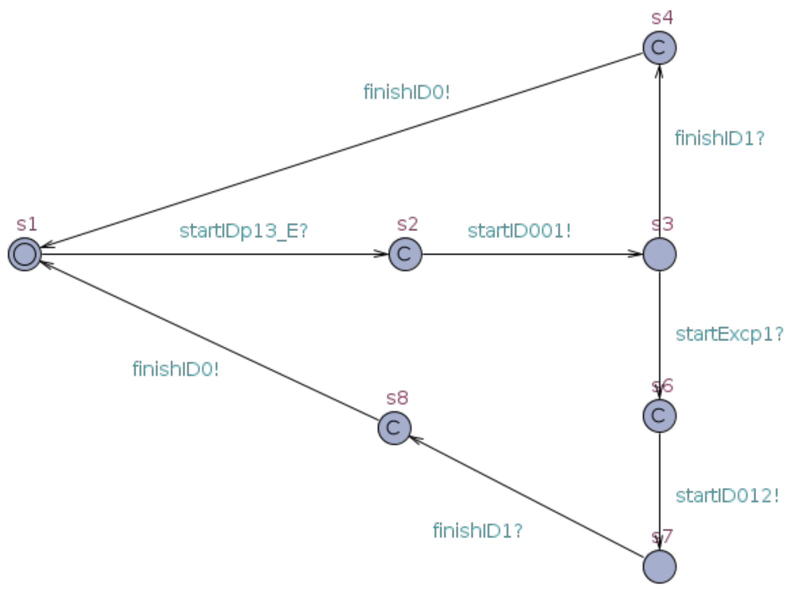
In Figure 38, we annotate the structure of the control TA for the operator exception with the names used in the translation Rule 3.14. The translated TA has eight locations and eight transitions defined in Lines 6–13 and Lines 14–22 respectively. Then, Lines 23–31 define the labels of the transitions. Lines 33–38 extracts and updates the list of names `excps`, which we used for handling exceptions in the tuples `usedNames`. Finally, Lines 41–44 define a recursive call for the subsequent translation of the remaining processes.

The behaviour of the control TA begins on transition `tran1` for performing a flow action. Then, on transition `tran2` the control TA performs another flow action that activates `Tp1`. After that, the control TA remains on location `loc3` until either `Tp1` terminates successfully or performs an exceptional action from the list `es`. If `Tp1` terminates with performing a termination action the translated TA synchronises with the corresponding co-action on transition `tran3`, and then performs another termination action on transition `tran4` for terminating the whole process.

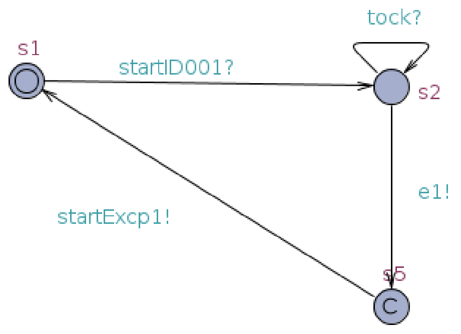
Alternatively, if `Tp1` performs an exception action that raises an exception action, the control TA synchronises with its co-action on transition `tran5`, and then immediately initiates the translation of `Tp2` on transition `tran6`, and then waits on location`loc7` until the translated list of TA `Tp2` performs a termination action and the control TA synchronises with the corresponding co-action on transition `tran7` and then on transition `tran8`, the control TA immediately performs another termination action for terminating the whole process. The following Example 3.17 illustrates using the Rule 3.14 in translating a process.

Example 3.17. An example that illustrates using Rule 3.14 in translating a process. This example translates the process `((e1->SKIP)[|{e1}|>(e2->SKIP))` into a list of TA as follows.

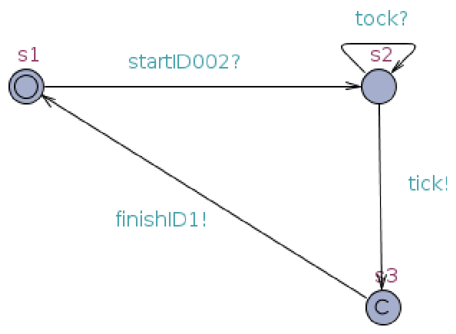
```
transTA((e1->SKIP)[|{e1}|>(e2->SKIP)) = [
```



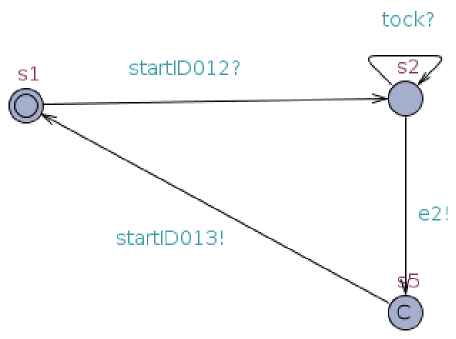
```
       ] ++ ta1 ++ ta2

where
ta1 = transTA(e1->SKIP)
    = [
```



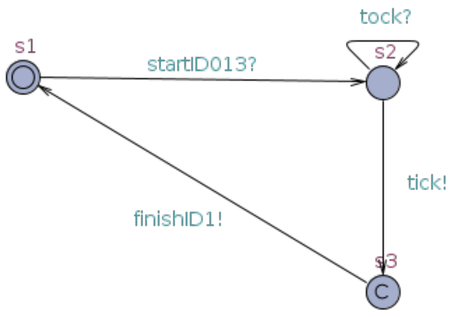
```
  ] ++ transTA(SKIP)
     = [
```

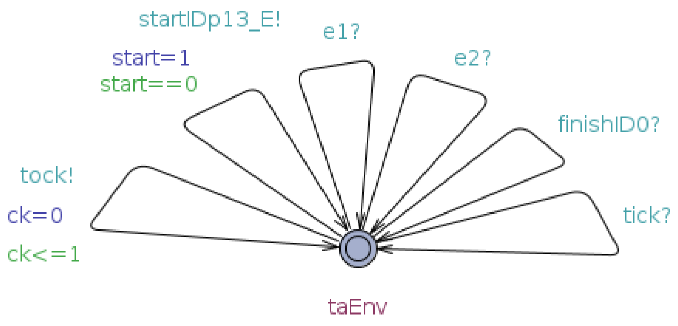


```
         ]

ta2 = transTA(e2->SKIP)
    = [
```



```
      ] = transTA(SKIP)
        = [
```





]

Example 3.17 translates the process `(e1->SKIP)[|{e1}|>(e2->SKIP)` into a list containing 5 TA. The first TA is a translation of the operator interrupt using Rule 3.14. Second and third TAs are translations of the LHS process `(e1->SKIP)`. Second TA captures the translation of the event `e1` using Rule 3.5. While the third TA captures the translation of the subsequent process `SKIP` using Rule 3.3. Similarly, the fourth and fifth TA are translations of the RHS process `(e2->SKIP)`. The fourth TA captures the translation of the event `e2` using Rule 3.5. The fifth TA captures the translation of the subsequent process `SKIP` using Rule 3.3.

The behaviour of the first TA begins with a flow action `startID13_E` that comes from the environment and then immediately performs the subsequent coordination action `startID001` which activates the second TA that initiates the behaviour of the LHS process `(e1->SKIP)`. After that, the first TA waits on location`s3` until it receives either a termination action `finishID1` or an exception action `startExcp1`.

If the first TA receives a termination action `finishID1` which indicates a successful termination of the LHS process `(e1->SKIP)`, then the first TA performs another subsequent termination action `finishID0` to signal a termination of the whole process. Alternatively, if the first TA receives an exception action `startExcp1`, then the first TA immediately performs the flow action `startID012` which activates the RHS process `(e2->SKIP)`. Then, the first TA waits on location`s7` until it receives a termination action `finishID1` and then immediately performs the subsequent termination action `finishID0` to signal the termination of the whole process. This completes the behaviour of the first TA for the translation of operator `Exception` in the process `(e1->SKIP)[|{e1}|>(e2->SKIP)`.

3.3.15. Translation of Timeout

This section describes the translation of the operator *Timeout*. The section begins with presenting a rule for translating the operator timeout and then follows with an example that illustrates using the rule in translating a process.

According to Roscoe [6], the operator timeout specifies a deadline for the LHS process to perform an event before the deadline or the process the RHS process begins its behaviour and the whole process behaves as the RHS process. In *tock-CSP*, this is express in term of internal choice and delay process as follows:

```
(P1 [d> P2 = P1) |~| (WAIT(2);P2)
```

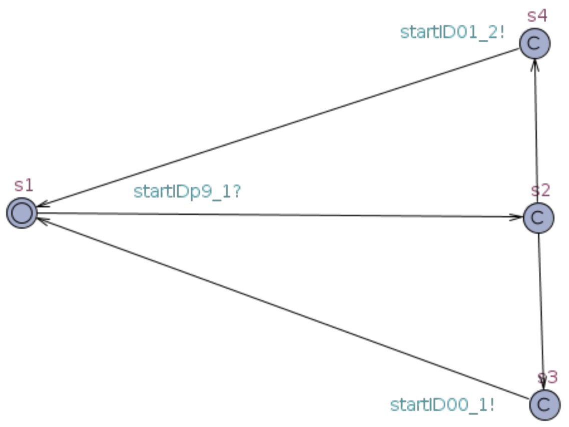
We follow a similar pattern in translating the operator timeout. We translate the operator in term of the two previous rules for translating internal choice (`|-|`) and a process delay `WAIT(d)`. Rule 3.15 expresses the translation rule and Example 3.18 demonstrates using the rule in translating a *tock-CSP* process.

Rule 3.15. Translation of Timeout

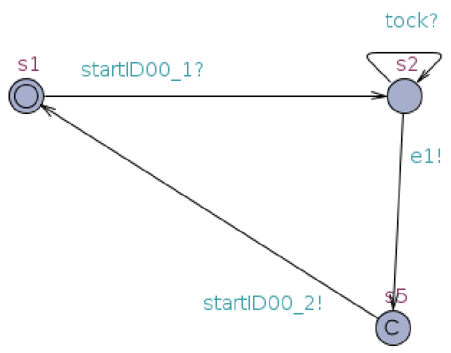
```
transTA (Timeout p1 p2 d) procName bid sid fid usedNames =
         transTA (IntChoice p1 (Seq (WAIT d) p2 )) procName bid
             sid fid usedNames
```

Example 3.18. An example that illustrates a translation of the process using Rule 3.14. This example translates the process `((e1->SKIP)[2>(e2->SKIP))` into a list of TA as follows.

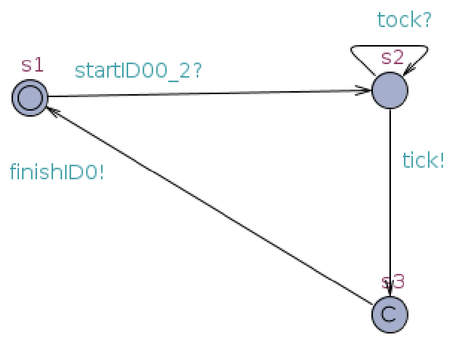
```
  transTA ((e1->SKIP)[2>(e2->SKIP)) =
    transTA ((e1->SKIP)|~|(WAIT(2);(e2->SKIP))) = [
```



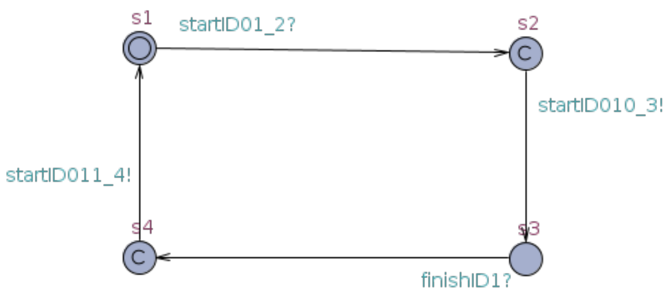
```
        ] ++ ta1 ++ ta2

where
ta1 = transTA(e1->SKIP)
    = [
```



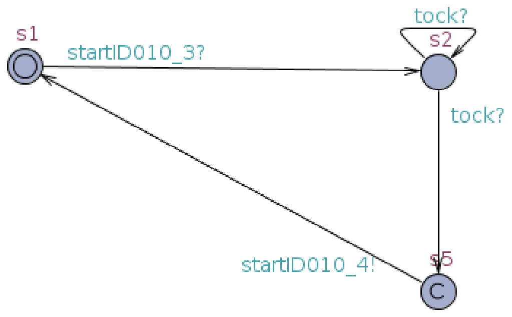
```
  ] ++ transTA(SKIP)
     = [
```



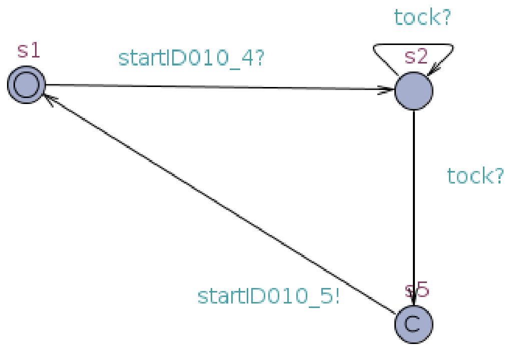
```
           ]

ta2 = transTA ((WAIT 2);(e2->SKIP))
    = [
```



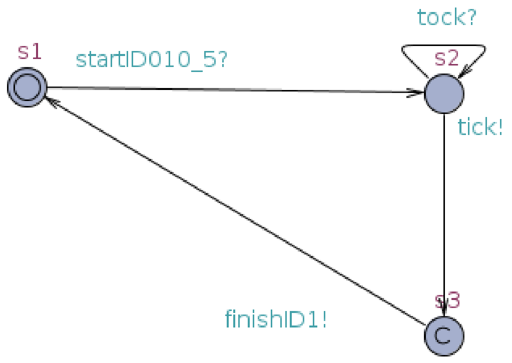
```
         ] = ta21 ++ ta22
ta21 = transTA(WAIT 2)
      = [
```



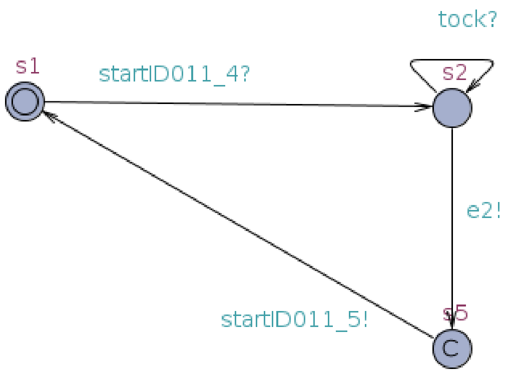
```
   ] ++ transTA(WAIT 1)
      = [
```



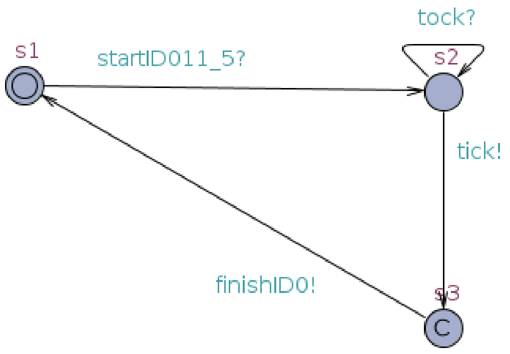
```
   ] ++ transTA(SKIP)
      = [
```

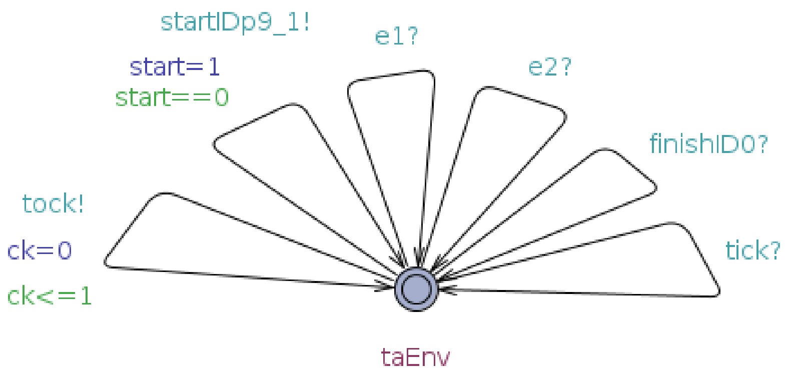


```
              ]

ta22 = transTA(e2->SKIP)
     = [
```



```
   ] ++ transTA(SKIP)
     = [
```





```
       ]
```

In Example 3.18, we illustrates using Rule 3.15 in the translation of the process ((e1->SKIP)[2>(e2->SKIP)) into a list of TA containing 10 TAs. The first TA is translation of the operator for internal choice. The second and third is translation of the LHS process (e1->SKIP). The second TA captures the translation of the event e1 according to Rule 3.5. The third TA captures the translation of the subsequent process SKIP. From the fourth to the ninth TA are translations of the RHS process (WAIT(2);(e2->SKIP)). The fourth TA is a translation of the operator sequential composition according to Rule 3.10. The fifth and sixth TA are translations of the delay process WAIT(2) according to Rule 3.6. The seventh TA captures the translation of the event e2 according to Rule 3.5. The eighth TA captures the translation of the subsequent process SKIP. The last TA is an environment TA for the list of translated TA of the process ((e1->SKIP)[2>(e2->SKIP)). This completes the description of an example that illustrates using Rule 3.15 in translating a process.

3.3.16. Translation of EDeadline (Event Deadline)

This section describes the translation of the construct Edeadline for a process that assigns a deadline to an event. The section begins with presenting a translation rule for the construct Edeadline and then follows with an example that illustrates using this rule in translating a process.

Rule 3.16 defines a translation of the construct Edeadline into a TA. In Figure 39, we annotate the structure of the TA with the names used in the translation rule. The TA has three locations and four transitions, as defined in Lines 6–9 and Lines 11–16 respectively. Lines 18 – 32 define the labels of the transitions. Line 24–25 defines a label for resetting the timer. Lines 27–28 update the time with one time unit after every action tock. Lines 30–31 define guards for controlling the deadline. The following example illustrates using this Rule 3.16 in translating a process.

Rule 3.16. Translation of EDeadline (Event Deadline)

```
transTA (EDeadline e n) procName bid sid fid usedNames =
  ([(TA idTA [] [] locs [] (Init loc1) trans)], [], [])
  where
    idTA = "taDeadln" ++ bid ++ show sid

    loc1 = Location "id1" "s1" EmptyLabel None
    loc2 = Location "id2" "s2" EmptyLabel None
    loc3 = Location "id3" "s3" EmptyLabel CommittedLoc
    locs = [loc1, loc2, loc3]

    tran1 = Transition loc1 loc2 ([lab1] ++ t_reset)    []
    tran2 = Transition loc2 loc2 ([lab2] ++
            dlguard ++ dlupdate) []
    tran3 = Transition loc2 loc3 ([lab3] ++ dlguard2)   []
    tran4 = Transition loc3 loc1 [lab4] []
    trans = [tran1, tran2, tran3, tran4] ++
            (transIntrpt intrpts loc1 loc2)

    lab1 = Sync (VariableID (startEvent procName bid sid) [])
        Ques
    lab2 = Sync (VariableID "tock"    []) Ques
    lab3 = Sync (VariableID (show e)  []) Excl
    lab4 = Sync (VariableID ("finishID" ++ show fid)   []) Excl

    -- reset timer
    t_reset = [(Update [(AssgExp (ExpID "tdeadline")
                                  ASSIGNMENT (Val 0))] )]

    dlupdate = [(Update [(AssgExp (ExpID "tdeadline")
                                 AddAssg (Val 1) ) ] ) ]

    dlguard  =
        [(Guard (BinaryExp (ExpID "tdeadline") Lth (Val n)))]
    dlguard2 =
        [(Guard (BinaryExp (ExpID "tdeadline") Lte (Val n)))]

    (_, _, _, _, _, intrpts, _, _) = usedNames
```

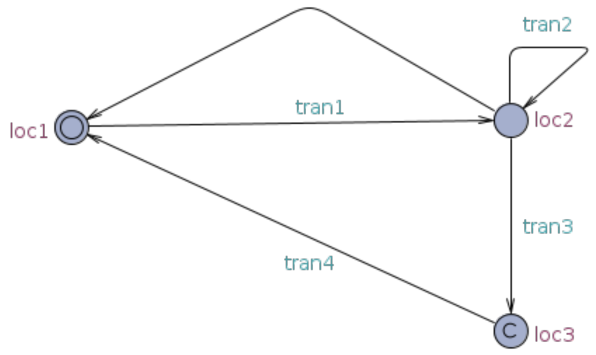
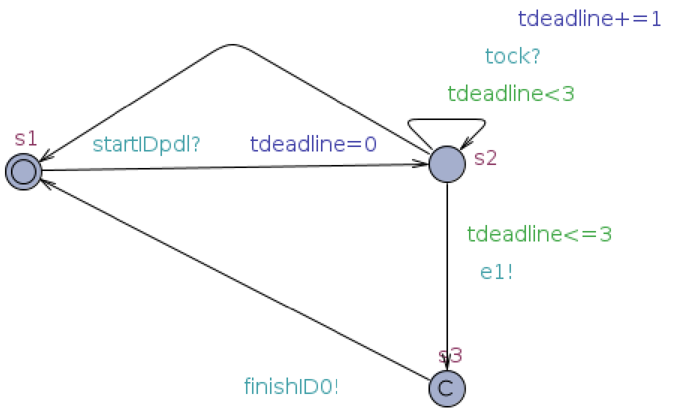
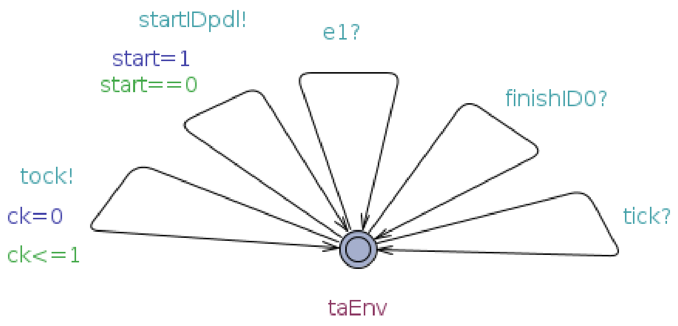


Figure 39: A structure of the control TA for the translation of the process Edeadline.

Example 3.19. This example illustrates using Rule 3.16 in translating the process `(EDeadline (e1, 3))` into a list of TA as follows.

```
 transTA (EDeadline (e1, 3)) "pdl" "0" 0 0
                         ([], [], [], [], [], [], [], ([],[]))
     = [
```





```
       ]
```

The behaviour of the first TA begins with synchronising on a flow action `startIDpd?` from the environment and then reset deadline `tdeadline` to zero. After that, on location `s2` either the TA performs the action `tock` to record the progress of time or the TA performs the event `e1` within a deadline of 3 time units. After the deadline `tdeadline` the guard `tdeadline<=3` blocks the event `e1` and the TA follows a silent transition to the initial location `s0`. The second TA is an environment TA for the list of translated TA. This completes the behaviour of the TA for the translation of the process `Edeadline(e1, 3)`.

3.3.17. Translation of Hiding

This section describes the translation of the operator for hiding events in the behaviour of a process. The section begins with presenting a rule for translating the operator `Hiding`, and then follows with an example that illustrates using the rule in translating a process.

Rule 3.17. Translation of Hiding

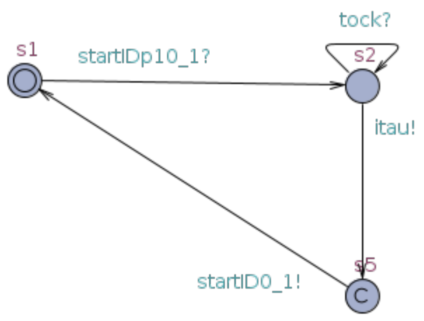
```
1 transTA (Hiding p es ) procName bid sid fid usedNames =
2         transTA p      procName bid sid fid usedNames'
3         where
4            (syncs, syncPoints, hides, renames, exChs, intrrs,
5                    iniIntrrs, excps) = usedNames
6
7            -- Updates the parameter for hiding
8            usedNames' = (syncs, syncPoints, (es ++ hides),
                renames, exChs, intrrs, iniIntrrs, excps)
```

Rule 3.17 updates the used name for hiding `hides`, which is used in the subsequent translations that are handled in Rule 3.5. Line 1 defines the function `transTA` for the construct `Hiding`. Line 2 describes the output in terms of the function `transTA` with an updated name `usedNames'`, which contains an updated name `hides`. Lines 4–5 extract the name `hides` from the tuples of used names `usedNames`. Lines 8–9 updates the name `usedNames` with hiding events for subsequent translation.

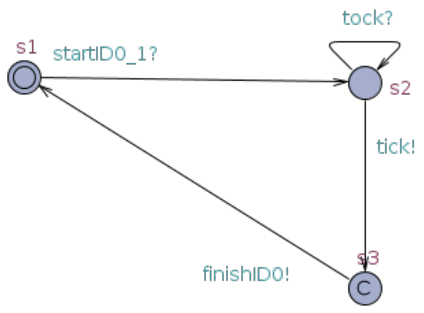
Rule 3.5 checks the used name `hides` in translating each event. If an event is in the list of hiding events, Rule 3.5 translates the event into a special name `itau`. While, if an event is not part of the used names `hides`, Rule 3.5 translates the event with its name in the output TA. The following Example 3.20 illustrates using this rule in translating a process.

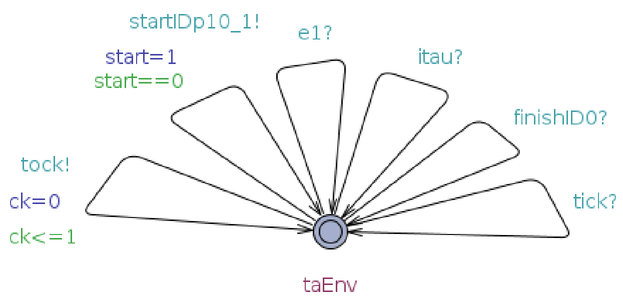
Example 3.20. This example demonstrates using Rule 3.17 in translating the process `((e1->SKIP)\{e1})` into a list of TA as follows.

```
1 transTA((e1->SKIP)\{e1}   "p10_1" _ 0 0 usedNames ) =
    transTA((e1->SKIP) "p10_1" _ 0 0 usedNames')
where
(syncs, syncMap, hides, rename, chs, intrpt, initIntrpt) =
    usedNames

usedNames' = (syncs, syncMap, [e1]++hides, rename,
                    chs, intrpt, initIntrpt)

transTA((e1->SKIP) "p10_1" _ 0 0 usedNames')
     = [
```



```
] ++ transTA(SKIP)
       = [
```





```
]
```

Example 3.20 illustrates a translation of a process using 3.17. The example translates the process `((e1->SKIP)\{e1})` into a list of TAs that contains three TAs. The first TA is a translation of the hiding event `e1` into a special name `itau`. The second TA is a translation of the subsequent process `SKIP` according to Rule 3.3. The third TA is an environment TA for the list of translated TAs.

The behaviour of the translated TA begins with the first TA, which synchronises on a flow action `startIDp10_1` from the environment TA, then performs the hiding action `itau`, and then immediately performs another flow action `startID0_1!` to activate the second TA. The second TA synchronises on the flow action `startID0_1?`, then performs the flow action `tick`, and then immediately performs the termination action `finishID0` that records a successful termination of the whole process. This completes the translation of the process `((e1->SKIP)\{e1})`.

3.3.18. Translation of Renaming

This section describes the translation of the operator Renaming. The section begins with presenting a rule for translating the operator renaming and then follows with an example that illustrates using the rule in translating a process.

Rule 3.18. Translation of Renaming

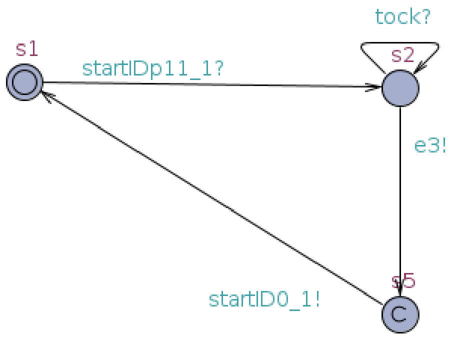
```
transTA (Rename p pes) procName bid sid fid usedNames
 =  transTA    p       procName bid sid fid usedNames'
    where
       (syncs, syncPoints, hides, renames, exChs, intrrs,
        iniIntrrs, excps) = usedNames

       -- Updates the name renames in the list of usedNames
       usedNames' = (syncs, syncPoints, hides, (renames ++ pes),
            exChs, intrrs, iniIntrrs, excps)
```

Translation of operator Renaming follows similar patterns with the previous Rule 3.17, except that, the parameter for renaming is a list of pairs, an event with its corresponding new name. This rule updates the used name `renames` from the tuples `usedNames`. Then in the subsequent translation, Rule 3.5 checks the updated used name `renames` in translating each event. If an event is in the list `renames`, Rule 3.5 replaces the event name with its corresponding new name, such that it appears with its new name in the translated TA.

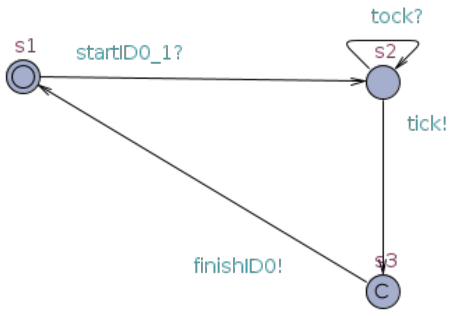
Example 3.21. An example for translating a process that demonstra a translation of the operator Renaming in translating the process `((e1->SKIP)[[e1<-e3]])`.

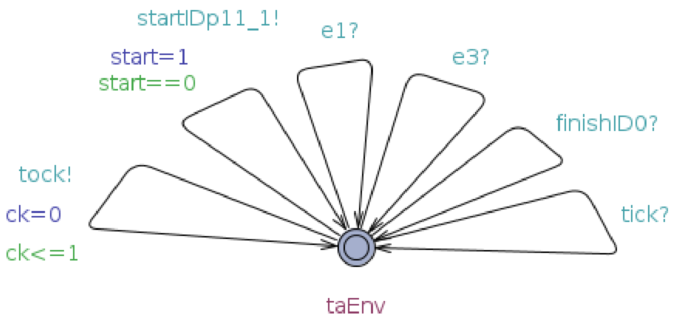
```
transTA(((e1-> SKIP)[[e1<-e3]]) "p11_1" [] 0 0 usedNames)
    = transTA((e1->SKIP) "p11_1" [] 0 0 usedNames')
```

```
where
    (syncs, syncMap, hides, rename, chs, intrpt, initIntrpt)
           = usedNames

    usedNames' = (syncs, syncMap, hides, rename ++ [e1, e3],
                        chs, intrpt, initIntrpt)
transTA((e1->SKIP) "p11_1" [] 0 0 usedNames')
    = [
```



```
   ] ++ = transTA(SKIP)
          = [
```





]

Example 3.21 illustrates using Rule 3.18 in translating a process into a list of TA that contains 3 TA. The first TA is a translation of the event `e1`, which appears with its new name `e3` in the translated TA. The second TA is a translation of the subsequent process `SKIP`. The last TA is an environment TA for the list of translated TA. This completes description of the translation of the process `((e1->SKIP)[[e1<-e3]])`.

3.3.19. Definition of Environment TA

This section defines an explicit environment TA for the translated TA of the Uppaal models. Definition 3.11 provides a Haskell function that expresses the structure of an explicit environment TA for the translated Uppaal system. Here we provide the definition of the environment TA, which we have seen various examples in the provided examples of the translation rules.

Definition 3.11. A Function for defining an environment TA

```
env :: String -> [Event] -> Template
env    pid       es      =
  Template "Env" [] [] [loc] [] (Init loc) trans
  where
   loc = Location  "taEnv" "taEnv" EmptyLabel None
   tll = Transition loc     loc
   -- a common name for defining list of transitions in the
       environment TA

   trans =
     [(tll [(Sync    (VariableID id []) Ques)] [])|(ID id)
        <- es] ++
     [ tll [(Sync    (VariableID "startID0_0"  []) Excl),
            (Guard   (BinaryExp (ExpID "start"   ) Equal (Val 0)
               )),
            (Update [(AssgExp   (ExpID "start"   ) ASSIGNMENT (
               Val 1))])] [],
     tll [(Sync    (VariableID ("startID" ++ pid) []) Excl),
        (Guard   (BinaryExp  (ExpID "start"   ) Equal (Val 0))),
          (Update [(AssgExp    (ExpID "start"   ) ASSIGNMENT (
             Val 1))  ]   ) ] [],
     tll [(Sync  (VariableID "finishID0" []) Ques)] [],
     tll [(Sync  (VariableID "tick"      []) Ques)] [],
     tll [(Sync  (VariableID "itau"      []) Ques)] [],
     tll [(Sync  (VariableID "tock"      []) Excl),
          (Guard (BinaryExp (ExpID "ck") Lte (Val 1))),
          (Update [AssgExp  (ExpID "ck") ASSIGNMENT (Val 0)])
          ] []
    ]
```

As highlighted in the translation strategy (Section 3.2), each process is translated into a list of TAs that includes an environment TA. The function defines the environment TA that has one location as defined in Line 5. While the remaining Lines 6–25 defines the transitions of the TA. First, Line 6 defines a common name used in defining the source and targets of all the transitions. Line 9 defines a transition for each action from the translated process. Lines 10–12 define the first starting flow action `startID0_0`, which initiates the behaviour of the translated TA (for the case of anonymous function). This starting transition has guards that block the environment from starting the behaviour multiple times. In the case of a named process, Lines 14–16 defines another transition for starting flow action that has the process name (for the case of a named process). Line 20 defines the final termination co-action for terminating the whole process. Line 21 defines a transition for a co-action of the translated action `tick`. Line 22 defines a transition for a co-action of hiding events `itau`. Lines 21–23 define a transition for the translated action `tock`, which is associated with a clock variable for recording the progress of time. This completes the definition of the environment TA. Also, this definition completes the presentation of the translation rules developed for translating *tock-CSP* into a list of TA.

3.4. Final Considerations

In this chapter, we discussed a technique for translating *tock-CSP* models into Uppaal models. The chapter begins with defining a BNF that served as the foundation of the translation work. The BNF describes the operators we considered for our targeted work. In essence, the developed BNF defines constructs for the selected operator of the *tock-CSP*.

Subsequently, we described the strategy we follow in achieving the translation work. Basically, we consider small size TAs joined together using additional flow actions, which we introduce in the translation work. We use the flow actions to coordinate the connections of the small TAs. The names of the flow actions were generated to be unique, and also provide a good structure for connecting the TA to form a network of TA. We consider using small sizes TAs in order to capture the compositional structure of *tock-CSP*. We used examples to illustrate using both the small TA and the flow actions that connect the translated TA.

Based on the developed translation strategy, we presented rules that precisely describe the translation of *tock-CSP* into TA. Each translation rule describes a translation of one construct of the BNF into TA. For precise description, we used Haskell notations for presenting the translation rules, where we defined a function transTA that defines a translation of each construct into TAs. The function `transTA` takes a *tock-CSP* model as an argument and provides an output list of TA that captures the behaviour of the input *tock-CSP* model. Examples were provided that illustrate using each rule in translating *tock-CSP* process. In the next chapter, we are going to discuss the evaluation of the translation technique.

Chapter Four

4. Evaluation

In this chapter, we describe the mechanisms we consider in evaluating the translation technique. In Section 4.1, we described a tool that implements the translation rules presented in Chapter 3. In Section 4.2, we describe another tool we developed for validating the translation tool. We use trace semantics for validating the translation technique. Thus, we develop a technique for generating and comparing traces of *tock-CSP* and TA.

Additionally, we use two forms of test cases for evaluating the translation tool, a collection of small processes and case studies. In Section 4.3, we describe a collection of small processes we used in assessing the correctness of the translation rules as part of the translation technique developed. In Section 4.4, we describe the second category of the test cases for evaluating the translation technique. Developing the translation technique give us a good opportunity for comparing the performance of the two model-checkers: FDR and Uppaal in analysing deadlock freedom, where the input *tock-CSP* process is constructed manually and the translated TA generated by our translation tool. Because of that, the results may not be a fair comparison, as the generated TA may not be the most efficient models for Uppaal. We describe the result of comparing the performance in Section 4.5. In Section 4.6, we describe a plan for mathematical proof of the translation technique. Finally, Section 4.7 provides a summary and conclusion of the chapter.

4.1. Mechanisation of the Translation Rules

In this section, we discuss a prototype tool we developed for automating the translation technique. After providing the translation rules, it is also useful to translate *tock-CSP* automatically, using the translation strategy to validate the translation technique. Besides, an automatic translation will enable considering the compatibility of the translation rules and other technical concerns. Thus, we create a tool for automatic translation based on our translation strategy (Chapter 3).

In implementing the translation tool, we continue using Haskell, the language we used previously in presenting the translation rules in Chapter 3, alongside using the Glasgow Haskell Compiler [116]. This reduces the complexity of combining the translation rules into a single system because the developed description of the translation rules in Haskell provides executable components of the translation tool. Figure 40 illustrates the structure of the translation tool. The implementation of the tool is available in the provided repository of the work [117].

Considering the functional structure of Haskell, we have developed another function `transform()` that encapsulates the translation rules `transTA()`, and the remaining two functions `syncTA()` and `envTA()` for synchronisation TA and environment TA, respectively (as defined in Section 3.3). The function `transform` uses the translation

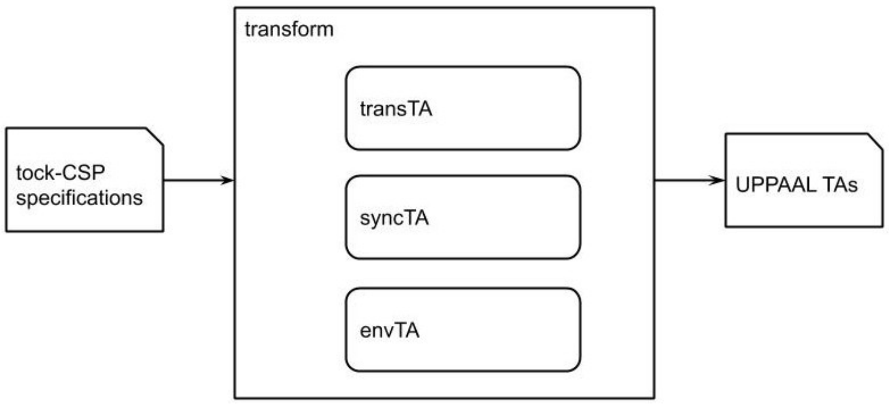


Figure 40: Structure of the function transform for the translation system

rules as part of the system that invokes the appropriate translation rule in translating each construct of the input *tock-CSP* specifications into the corresponding translated TA. Then, finally the function `transform` completes the translation with providing both synchronisation TA and an environment TA that closes the translated system, as described in Chapter 3.

In using the translation tool, the function `transform()` (Section 3.2) takes a valid name *tock-CSP* process as input Abstract Syntax Trees (AST), and initialises the required arguments for the three functions: `transTA`, `syncTA` and `envTA`. The input of the translation tool is a valid *tock-CSP* specification within the scope of this work.

The output of the function `transform()` is a network of TA for Uppaal, which is based on a provided Uppaal template for the internal representations of the networked TA, as shown in Figure 41. In this work, we encode the template as a data structure in Haskell, which captures the output of the translation technique into a suitable Uppaal TA. The definition of template TA has five major sections: header, declaration, network of TA, system definition and TCTL queries, as shown in Figure 41. First, the header section describes the configuration information of the translated TA. Second, a declaration section defines the terms used in the system. Third, a list of definitions for the list of TAs that form the networked TA. Fourth, the system definition instantiates the definitions of the TA into a system. Lastly, a list of queries describes requirement specifications of the system. An additional detailed structure of Uppaal TA is provided in Appendix C.

We provide an interface for running the translation tool, which has a list of commands for accessing the tool. The tool provides an interactive environment for running the translation tool, which accepts an input *tock-CSP*, either directly as input to the command prompt or uploading a *tock-CSP* file that contains specifications (in AST) of a system. This completes the description of the translation mechanism. In the next section, we will discuss the technique consider in generating and comparing the traces of *tock-CSP* and its corresponding translated TA.

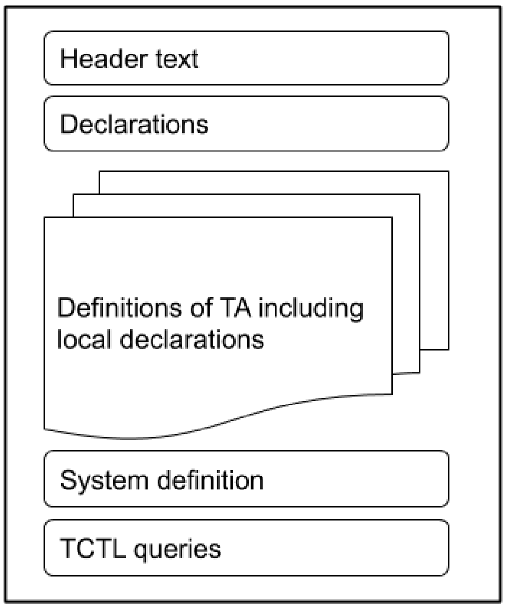


Figure 41: Structure of the Uppaal models

4.2. Trace Analysis

This section discusses the mechanism we develop for evaluating the translation rules. A sound translation preserves properties of the source model. This is determined by comparing the behaviours of the source model and the translated model [118–121]. In this work, we use traces to compare the behaviour of the input *tock-CSP* models and the traces of the translated TA models.

We have developed another technique along with its software tool, which automates the validation technique using trace analysis. The structure of the trace analysis tool is in Figure 42, which has five components, a controller, a translation system (Section 4.1), FDR and Uppaal as black-boxes, and two additional components for analysing traces in two stages. The tool uses the translation tool in translating *tock-CSP* model into TA, and then uses both FDR and Uppaal as black boxes in generating sets of finite traces. Then, the tool compares the generated traces in two stages, 1st and 2nd stage.

The benefit of adding the 1st stage is because it is easier to generate the traces with the 1st stage, unless for concurrency in which the 1st stage generates incomplete traces of a process that has concurrency, due the complexity of using Uppaal in generating traces of concurrent processes. Therefore, we create the second stage to complement the 1st stage with the additional technique for using the power of FDR to generate traces of a process that contains concurrency. The system begins with the 1st stage if the traces are complete the system terminates. Otherwise, the system invokes the 2nd stage for computing the remaining traces.

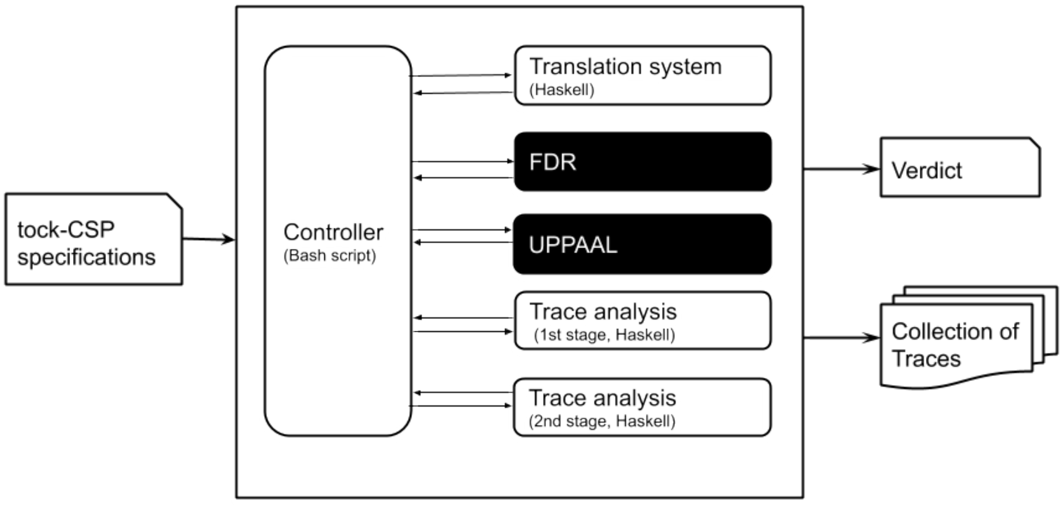


Figure 42: Structure of the trace analysis system

In Figure 42, the controller connects the components of the trace analysis system, which invokes each component of the system one after the other, passing input and collecting output. The controller passes the required input to a component and collects output and then proceeds to pass the collected output as input to the subsequent component of the system for further analysis. We used two stages of analysing traces. In the first stage, we generate traces from both FDR and Uppaal. If the traces do not match, we develop the second stage, where we use the power of FDR to complement Uppaal in analysing the traces.

1st stage of trace analysis We describe the steps of the first stage as follows:

Step 1: The controller takes input specification *tock-CSP*.

Step 2: Invokes the translation system to get the translated TA.

Step 3: Invokes FDR to generate traces of the input *tock-CSP*.

Step 4: Invokes Uppaal to generate traces of the translated TA.

Step 5: Compares the generated traces (1st stage of the trace analysis).

Step 6: If the generated traces from FDR and Uppaal do not match, we create a second stage of the trace analysis, where we check if all the generated traces of *tock-CSP* are valid traces of the translated TA.

In the first stage of the traces analysis, we compare the generated traces of *tock-CSP* and TA. However, due to the nature of using TCTL language in formulating queries for generating the traces, it is impossible to retrieve all the possible traces of TA, especially for the case of TA that captures concurrency. To address this complexity, we develop the second stage of the trace analysis, where we use the power of FDR to complement UPPAAL in generating traces. Thus, we verify all the generated traces of *tock-CSP* are valid traces of the translated TA, which validates the equivalence of the traces. Details of the second stage are as follows. As a running example for the second stage, we consider the following *tock-CSP* process.

```
P1 = e1->((e2->SKIP)|||(e3->SKIP))
```

2nd Stage of the trace analysis: steps of the second stage are as follows:

Step 1: The second stage of the technique begins with appending a special event `mark` to the process `P` to form another process `Pm`. This is express as follows:

```
Pm = P;(mark -> STOP)
```

Note: The selected special event `mark` must not be part of the process `P`.

Step 2: We create an auxiliary process that specifies the required length `n` of a trace. The process is expressed as follows.

```
Sl(n) = if n == 0 then (mark -> SKIP)
        else ([]ev : Events @ ev -> Sl(n-1))
```

The process $Sn(n)$ controls the size of a trace, such that either the process terminates after reaching a trace of size n or the process terminates after performing its last event. When $n == 0$, the process terminates after performing the special event `mark`; otherwise the process proceeds to perform any event from the set `Events` and decreases the value of the parameter `n` by 1, until $n == 0$.

Step 3: We put the process `Sn(n)` in parallel with the constructed process in Step 1. For the running example, the process is as follows:

```
Pmn = (Pm [|Events|] Sl(n))
```

In this case, either the process `Pm` terminates first or the process `Sl(n)` terminates after reaching the required length n of a trace, which forces the concurrent process `Pmn` to a deadlock after reaching the target length n for the required length of the traces.

Step 4: Then, we use FDR to verify an assertion `Pmn` refines `P`, which checks if the process `P` contains the traces of `Pmn`, as follows.

```
assert P [T= Pmn
```

For the running example, using the process `P` with $n > 3$, the assertion fails and yields a counterexample as follows:

```
t1 = <e1, e2, e3, m>
```

The counterexample is one of the complete traces of process `P` with the addition of the special event m. This is because of the formulated process `Pmn` performs the additional special event `mark`, which the process P cannot perform. In the case of a non-terminating process, the function `Sl(n)` forces the process to terminate after reaching the target length `n`. Then, we proceed to find out if the process `P` has another trace different from the generated trace `t1`.

Step 5 We convert the generated trace into a linear process.
For the running example, we convert the trace `t1` into the following linear process:

```
Pt1 = e1->e2->e3->mark->SKIP
```

Step 6: We use external choice to compose original process P with the constructed process Pt1; as `P[]Pt1`. Then, we use FDR again to check if the constructed process `P[]Pt1` contains the traces of the process `Pmn`, as follows.

```
(P[]Pt1) [T= Pmn
```

In the running example, the assertion fails and generates another counterexample, as follows:

```
t2 = <e1, e3, e2, mark>
```

Loop : We repeat Steps 5 and 6 until the assertion passes.
In the case of this running example, the assertion fails. Therefore, we execute the loop again and we repeat Steps 5 and 6.

Repeating Step 5: We convert the new trace `t2` into another linear process.

```
Pt2 = e1->e3->e2->mark->SKIP
```

Repeating Step 6: We use external choice to compose the new process `Pt2` with the previously formulated process `(P[]Pt1)` to form another process `(P[]Pt1[]Pt2)`. Then, we use FDR to verify if the new process `(P[]Pt1[]Pt2)` contains the traces of the process `Pm`, as follows.

```
(P[]Pt1[]Pt2) [T= Pmn
```

After repeating Step 6, the assertion passes, which indicates that the traces t1 and t2 are the only traces of length 3 for the input process

```
P1 = e1->(e2->SKIP)|||(e3->SKIP)
```

In generating multiple traces with FDR, like most of the model-checking tools, FDR was developed to produce only one trace (counterexample) at a time. As such, based on the testing technique developed in [122], we develop a technique that repeatedly invokes FDR for generating traces until we get the required traces. We used the developed technique in generating finite length of traces for the input *tock-CSP* specifications using FDR.

However, this technique for generating traces using FDR is not suitable for generating traces using Uppaal. This is due to the differences between the two systems. FDR was developed based on verification using refinement. Secondly, FDR uses the same language CSP for both specifications and verification of a system, while Uppaal was developed based on networks of TA. Also, Uppaal uses different languages for specification and verification; Uppaal uses TA for modelling the specifications of a system and TCTL for specifying the verification requirements.

Thus, we developed another technique for generating traces of TA. The technique we developed is based on a testing technique developed for Uppaal [123]. The basic idea of the testing technique [123, 124] was described using Uppaal as black-box in generating test-cases that achieve test coverage criterion in TA. The testing technique was developed to traverse every edge in TA to achieve edge coverage. The edge coverage criterion was formulated using reachability property with additional auxiliary variables e1...en of types Boolean; each path has a unique combination of these variables es. On traversing each path, we assign the value true to all the auxiliary variables in the path, which facilitates formulating another test case for another path that has a different combination of the auxiliary variables. In the end, exhausting all the possible paths provides a set of test cases that achieves test-coverage criterion in a network of TA that models a system [123, 124].

In generating the traces, we attach two auxiliary variables to each action that captures a translation of an event. The two variables are path identification variable and depth value, epn and dp, respectively, as shown in Figures 44, 45 and 47 (Example 4.1). The three TA capture the occurrence of the actions e1, e2 and e3.

Initially, the auxiliary variables of the forms epx are initialised to the value false, then traversing a path and capturing its trace assigns the value true to all the auxiliary variables in that path. While the second auxiliary variable dp is initialised to the value 0, traversing each action increases the depth dp with the value 1, which increases the length of the trace. This technique enables us to capture traces of TA within the required depth. We illustrate the technique in the following steps.

Example 4.1. Consider a running example of a system that performs an action e1 and then performs two additional actions e2 and e3, concurrently. We translate the system into a network of TA in Figures 44 – 49.

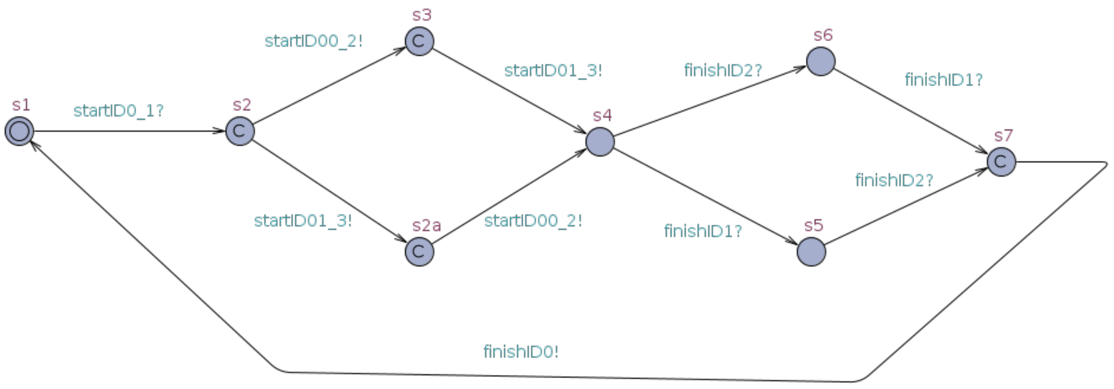


Figure 43: TA1 shows the translation of concurrency operator.

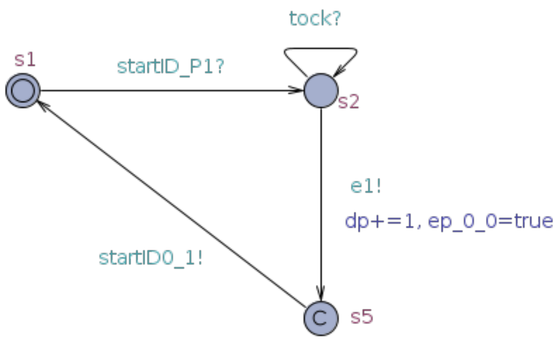


Figure 44: TA2 show the translation of the action e1, with the added auxiliary variables dp and ep_0_0

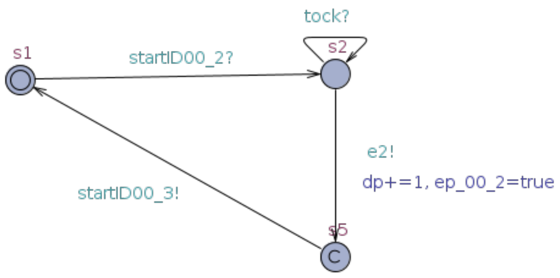


Figure 45: TA3 for translating the action e2 with the auxiliary variables

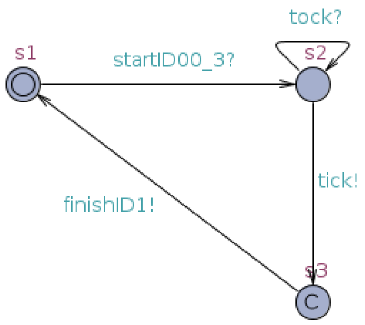


Figure 46: TA4 captures the termination process SKIP

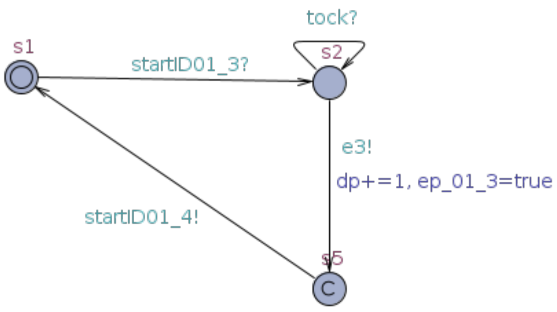


Figure 47: TA5 for translating the action e3 with auxiliary variables

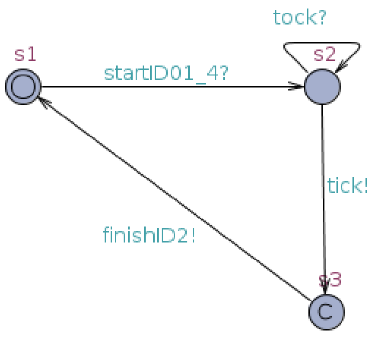


Figure 48: TA6 for translating the termination process SKIP

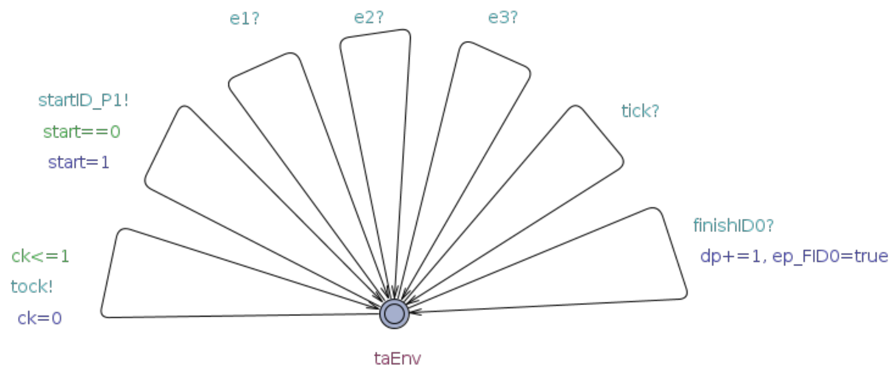


Figure 49: TA7 for an explicit environment of the translated TA

Step 1: We formulate a query for a requirement that the system should reach depth n for a trace of size n, as follows:

```
E<> (dp==n)
```

For the running example, we formulate a query to check that the system should reach a trace of length 3, expressed as follows:

```
E<> (dp==3)
```

Step 2: If the query passes, we get a trace $t1$ for path $p1$ from the initial state to the depth n. The path $p1$ is characterised with a list of the added auxiliary variables `e1...en`, such that all the auxiliary variables in the path are set to the value *true* on traversing the path.

For the running example, the query passes and returns the first trace

```
t1=[e1, e2, e3]
```

Also we characterise the path with a list of auxiliary variables

```
[ep_0_0, ep_00_2, ep_01_3]
```

that are set to true on generating the trace $t1$ in that path.

Step 3: We update the query by blocking the path $p1$, expressed as follows:

```
E<> ((dp == n) and  not p1)
```

Where `p1` is a conjunction of the list of auxiliary variables that identify the path `p1`, as in the case of this running example, we update the query by blocking the path `p1`, as follows:

```
E<> ((dp == 3) and not (ep_0_0 and ep_00_2 and ep_01_3))
```

Loop: We repeat steps 2 – 3 until the query fails.

Repeating Step 2: For the running example, checking for an alternative trace with the second query fails. This indicates that there is no alternative path for generating another trace. If the second query passes, we will get another trace *t*2 for an alternative path *p*2.

NOTE: However, there is another alternative trace `t2 = [e1, e3, e2]` that can be generated on traversing another path `[ep_0_0, ep_01_3, ep_00_2]`. However, it can be seen that the description of the two paths evaluate to the same logical value; even though they have a different order of the actions, which makes it difficult for Uppaal to detect the second trace [13]. As a result of that, we develop Stage 2 for analysing the traces, which we used in generating traces that are difficult to generate in Uppaal.

This is the main reason for generating traces in two stages. In stage 1, we generate traces using both TA and *tock-CSP* as well as using both FDR and Uppaal, respectively. Then, we compare the traces if the traces do not match, we move to Stage 2. Since all the traces of *tock-CSP* have precise order in Stage 2, we use the traces in guiding Uppaal to check if all the traces produced with FDR are valid traces of the translated TA. Details of Stage 2 are as follows.

We proceed to Stage 2 for checking if all the traces of the input *tock-CSP* are valid traces of the translated TA. We continue using the previous running example for the list of the translated TA in Figures 44 – 49. Previously, we generated the trace $t1 = \langle e1, e2, e3\rangle$ for the TA model using Uppaal. Also, in describing the technique for generating traces of *tock-CSP* using FDR, we generated the two traces $\{\langle e1, e2, e3\rangle, \langle e1, e3, e2\rangle\}$ for the input *tock-CSP* model.

Trace analysis Stage 2: the steps for the trace second stage of the trace analysis are as follows:

Step 1: We take the difference df between the two lists of traces.

```
df = (traces of tock-CSP) - (traces of TA)
```

For the running example, the difference between the two traces is:

```
df = [[e1, e2, e3], [e1, e3, e2]] - [[e1, e2, e3]]
```

```
∴ df = [[e1, e3, e2]]
```

[13]Perhaps, there might be another way of using the operator or other similar construct. This is left as part of the future investigation.

Step 2: For each trace *ti* in the traces df, we convert the trace *ti* into a linear TA $tiTA$, which has a special final state fs.

For the running example, we convert the trace [e1, e3, e2] into TA in Figure 50.

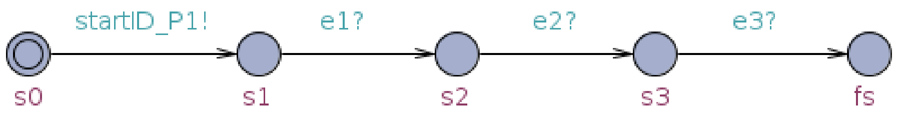


Figure 50: $tiTA$ - a linear TA for the trace [e1, e3, e2]

Step 3: In the list of translated TA, we replace the environment TA with the linear TA $tiTA$.

Therefore, for the running example in Figures 44 – 49, we replace the environment TA Figure 49 with $tiTA$ Figure 50.

Step 4: We formulate a query to check if the system reaches the final state fs of the trace TA $tiTA$ (after replacing the environment TA).

For the running example, the query is as follows:

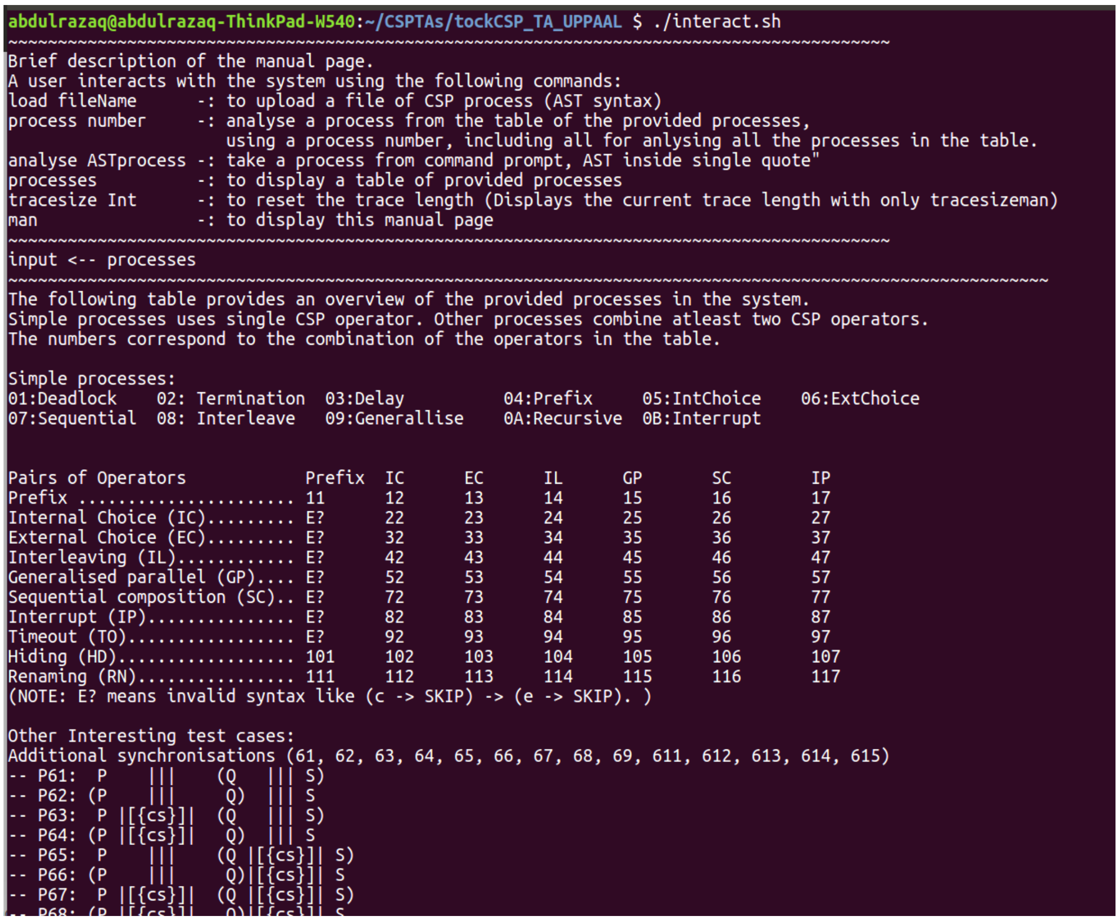
```
E <> fs
```

Step 5: If the query passes and the system finds a path to the final state fs of the TA $tiTA$, then the trace *ti* is a valid trace of the system. Otherwise, if the query fails, then there is no path to the final state of the $tiTA$. Therefore, the trace *ti* is not an acceptable trace of the translated TA.

In the case of the running example, the query passes. This indicates that the trace *ti* is a valid trace of the system. There is no other trace apart from the trace *ti*, because the difference between the two traces is *ti* only. Therefore, we complete the trace analysis.

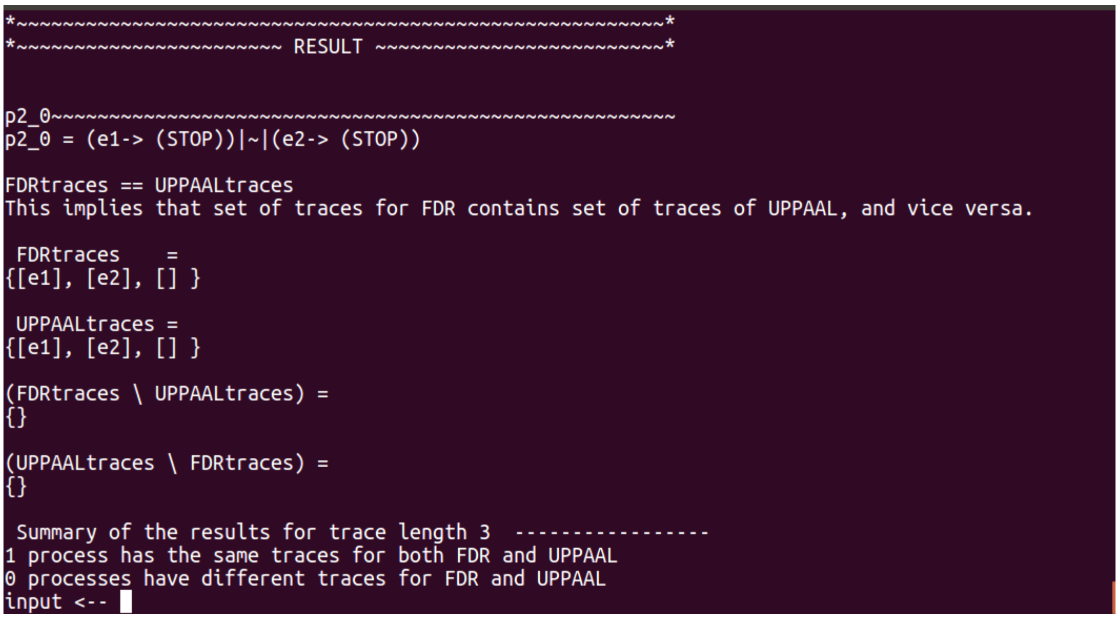
This completes the descriptions of the techniques we used for extracting the traces of both *tock-CSP* and TA, using both FDR and Uppaal. A prototype implementation of the trace analysis tool is available in the repository [117]. The tool has an interface that enables users to interact with the tool. We provide examples inside the prototype system that help users to understand the system. Figure 51 illustrates part of the interface, which displays the available commands for interacting with the tool.

The interface shows a brief description of the available commands. The first command, `load Filename` reads a file that contains *tock-CSP* specifications in AST. The second command `process number` analyses an existing process from the provided processes inside the system. In Figure 51, below the list of commands, there is a table of the provided processes with their corresponding numbers. The numbers are used for invoking each process from the interface. Figure 52 shows an example of analysing process number 20, `p2_0 = (e1->(STOP))|~|(e2->(STOP))}`, which illustrates a translation of the construct internal choice. The third command `analyse ASTprocess` analyses a process (specified in AST). The fourth command



```
abdulrazaq@abdulrazaq-ThinkPad-W540:~/CSPTAs/tockCSP_TA_UPPAAL $ ./interact.sh
~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
Brief description of the manual page.
A user interacts with the system using the following commands:
load fileName      -: to upload a file of CSP process (AST syntax)
process number     -: analyse a process from the table of the provided processes,
                      using a process number, including all for anlysing all the processes in the table.
analyse ASTprocess -: take a process from command prompt, AST inside single quote"
processes          -: to display a table of provided processes
tracesize Int      -: to reset the trace length (Displays the current trace length with only tracesizeman)
man                -: to display this manual page
~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
input <-- processes
~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
The following table provides an overview of the provided processes in the system.
Simple processes uses single CSP operator. Other processes combine atleast two CSP operators.
The numbers correspond to the combination of the operators in the table.

Simple processes:
01:Deadlock    02: Termination  03:Delay          04:Prefix     05:IntChoice    06:ExtChoice
07:Sequential  08: Interleave   09:Generallise    0A:Recursive  0B:Interrupt


Pairs of Operators            Prefix  IC      EC      IL      GP       SC        IP
Prefix ...................... 11      12      13      14      15       16        17
Internal Choice (IC)......... E?      22      23      24      25       26        27
External Choice (EC)......... E?      32      33      34      35       36        37
Interleaving (IL)............ E?      42      43      44      45       46        47
Generalised parallel (GP).... E?      52      53      54      55       56        57
Sequential composition (SC).. E?      72      73      74      75       76        77
Interrupt (IP)............... E?      82      83      84      85       86        87
Timeout (TO)................. E?      92      93      94      95       96        97
Hiding (HD).................. 101     102     103     104     105      106       107
Renaming (RN)................ 111     112     113     114     115      116       117
(NOTE: E? means invalid syntax like (c -> SKIP) -> (e -> SKIP). )

Other Interesting test cases:
Additional synchronisations (61, 62, 63, 64, 65, 66, 67, 68, 69, 611, 612, 613, 614, 615)
-- P61:  P    |||     (Q   ||| S)
-- P62: (P    |||      Q)  ||| S
-- P63:  P |[{cs}]|  (Q   ||| S)
-- P64: (P |[{cs}]|   Q)  ||| S
-- P65:  P    |||     (Q |[{cs}]| S)
-- P66: (P    |||      Q)|[{cs}]| S
-- P67:  P |[{cs}]|  (Q |[{cs}]| S)
-- P68: (P |[{cs}]|   Q)|[{cs}]| S
```

Figure 51: Interface of the trace analysis tool



```
*~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~*
*~~~~~~~~~~~~~~~~~~~~~~~ RESULT ~~~~~~~~~~~~~~~~~~~~~~~~~*


p2_0~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
p2_0 = (e1-> (STOP))|~|(e2-> (STOP))

FDRtraces == UPPAALtraces
This implies that set of traces for FDR contains set of traces of UPPAAL, and vice versa.

 FDRtraces    =
{[e1], [e2], [] }

 UPPAALtraces =
{[e1], [e2], [] }

(FDRtraces \ UPPAALtraces) =
{}

(UPPAALtraces \ FDRtraces) =
{}

 Summary of the results for trace length 3  ----------------
1 process has the same traces for both FDR and UPPAAL
0 processes have different traces for FDR and UPPAAL
input <--
```

Figure 52: An example of using the trace analysis tool for comparing the traces.

`processes` displays a table of the provided processes in the system, as shown in Figure 51, below the list of commands. The fifth command `trace size` sets the required depth of traces for the analysis. Lastly, the command `man` displays the manual page of the tool.

Figure 52 shows a sample output of the trace analysis system, which displays a summary of comparing the generated traces with both FDR and Uppaal. Also, the system generates a folder named `genFiles`, which contains files for the details of all the generated traces of both FDR and Uppaal as well as the details of comparing the generated traces. This concludes the description of the tool we developed for analysing and comparing the traces of the input *tock-CSP* and its corresponding translated TA.

4.3. Experimental Evaluations

This section discusses the list of processes we used in evaluating the translation technique. In evaluating the translation technique, we consider an experimental approach, which enables us to use the trace semantics in justifying the correctness of the translation technique. We begin with formulating a list of processes that covers interesting conditions (test cases), ranging from basic processes to a list of processes that pair all the *tock-CSP* constructors in the provided BNF (Section 3.1).

After formulating the processes, we use the translation tool (see Section 4.1) to translate the *tock-CSP* processes into TA. Then, we used the traces analysis tool (see Section 4.2) to generate and compare the traces of these formulated processes. The results of comparing and analysing the traces enable us to reason about the correctness of the translation technique. For each of the finite processes within a specific length, we consider traces of length 1, 2, 3 and length 10. We select these lengths as samples because we will not be able to cover all the possible lengths of the traces. In all the traces, the traces of the translated TAs model contain the traces of the original input *tock-CSP* process. The result justifies that the translated TA captures the behaviour of the input *tock-CSP* model.

Additionally, these formulated processes are part of the built-in processes provided in the translation tool. These provided processes help users to explore and understand the tool. Each process has a number, and we provide an interface for selecting each process using the provided processes number (additional details in the appendices). On entering a process number, the system displays the selected process, translates the process into TA, then generates and compares the traces of the selected process. Subsequently, the system displays the summarised results for comparing the traces. The list of provided built-in processes and the interface provide a good starting point for interacting and understanding the system.

4.4. Example - Case Studies

In this section, we discuss a list of case studies that illustrate the application of the translation technique. In selecting the case studies, we consider the wider definition of robot in a wider sense as described by IEEE Standard 1872-2015 [125] From the literature, we consider six cases: a cash machine (ATM) [126], a modified version of the cash machine (ATM2) (Section 4.4.1), an automated barrier to a car park [37] (Section 4.4.2), a railway crossing system [6] (Section 4.4.3),a thermostat machine for monitoring ambient temperature [37] (Section 4.4.4) and a simple mobile system [127] (Section 4.4.5).

First, an overview of these selected case studies is provided in Table 5, while detailed specifications of the system are described in the following Sections. Second, we write *tock-CSP* specifications for the case studies, as listed below for each system. Third, we use the translation tool in translating the *tock-CSP* specifications into TA for Uppaal. Lastly, we use Uppaal for verifying the sample properties listed under the verification requirements that follow each of the case studies.

4.4.1. Cash Machine (ATM)

This example illustrates a translation of a cash machine that goes through cycles of accepting a card, requiring its PIN, and servicing one request before returning the card to the customer. The request can be cash withdrawal, transferring cash and checking a balance of the account. If the PIN is incorrect, the machine returns the card and continues with an operation that prepares the machine for accepting another card [126].

In Listing 2, we present the specification of the system cash machine in *tock-CSP*. Line 1 defines the channels used in describing the system. Line 3 defines the time required for each event. We assign 0 to indicate that each event happens instantly and takes no time to complete. Line 5 describes the beginning of the timed section

Case Studies	**Tool**	**Average Timing (second)**	**States**	**Transitions**	**Events**	**States per Transitions**
Thermostat	FDR	0.0052	7	16	5	0.4375
	UPPAAL	0.0106				
Bookshop Payment	FDR	0.0048	7	32	9	0.21875
	UPPAAL	0.0011				
Simple ATM	FDR	0.0051	15	33	15	0.4545
	UPPAAL	0.0084				
AutoBarrier	FDR	0.0059	35	84	10	0.4167
	UPPAAL	0.0134				
Rail Crossing	FDR	0.0054	80	361	12	0.2216
	UPPAAL	0.0072				

Table 5: Timing of the selected test cases with an overview of their structure.

in a CSP file for FDR verification [41]. Lines 7–8 define the process `ATM` that checks valid PIN. The process accepts the card and PIN and then internally decides whether the PIN is valid and behaves as the process `Options`. For invalid PIN the process returns the card and waits for another card. Lines 9–13 define the behaviour of the process `Options`, which provides three options: `withdrawCash`, `checkBalance` and `transfer`. The first option `withdrawCash` takes an amount of the required money, dispenses the required cash, returns the card and behaves as the process `ATM` (waiting for another card). The second option `checkBalance` displays the account balance, returns the card and behaves as the process `ATM`. Last option `transfer` accepts both an account number and amount for the transfer; it then returns the card and behaves as the process `ATM`. Finally, Line 15 describes an assertion for checking deadlock freedom in the specifications. This completes the specifications of the system cash machine in *tock-CSP*.

Verification requirements: after using our translation technique to translate *tock-CSP* into TA, the following sample requirements are specified with TCTL and verified with UPPAAL automatically.

1. $A\Diamond$ `returnCard`
   The machine eventually returns the accepted card.

Listing 2: Specifications of Cash Machine (ATM)

```
channel tock, card, acceptPIN, correctPIN, incorrectPIN, returnCard,
    withdrawCash, amount, dispenseCash, returnCArd, checkBalance,
    displayBalance, transfer, acceptAccountNo, acceptAmount

OneStep(_) = 0

Timed(OneStep){

ATM = card -> acceptPIN -> (correctPIN ->  Options)
                        |~| (incorrectPIN -> returnCard -> ATM)
Options =
     (withdrawCash -> amount -> dispenseCash -> returnCard -> ATM)
     [] (checkBalance -> displayBalance -> returnCard  -> ATM)
     [] (transfer -> acceptAccountNo -> acceptAmount ->
         returnCard -> ATM)

assert ATM :[deadlock-free]
}
```

2. `withdrawCash --> returnCard`
   After withdrawing cash, eventually the machine will return the card.

3. `checkBalance --> returnCard`
   After checking account balance, eventually the machine will return the card.

4. `transfer --> returnCard`
   Whenever the machine accepts a card, eventually the machine will return the accepted card.

In a modified version in Listing 3, we extend the previous version of the cash machine that enables us to formulate additional interesting requirements that we are unable to specified in the previous version of the models. There are two modifications. First, the system retains the card after 5 seconds if the user did not take the card. Second modification, before dispensing the money, the system checks if there is enough balance for the requested amount to withdraw.

Verification requirements: after using our translation technique to translate *tock-CSP* into TA, the following sample requirements are specified with TCTL and verified with UPPAAL, automatically.

1. $A\diamond$ `retainCard`
   The machine eventually retains the card (expected to fail).

Listing 3: Specifications of Modified Cash Machine (ATM2)

```
channel tock, card, acceptPIN, correctPIN, incorrectPIN, returnCard,
    withdrawCash, amount, dispenseCash, returnCArd, checkBalance,
    displayBalance, transfer, acceptAccountNo, acceptAmount,
    retainCard, insufficientBalance, enoughBalance, takeCard

OneStep(_) = 0

Timed(OneStep){

ATM = card -> acceptPIN -> (correctPIN ->  Options)
                           |~| (incorrectPIN -> ReturnCard)

Options = (withdrawCash -> Withdrawal)
          [] (checkBalance -> displayBalance -> ReturnCard)
          [] (transfer -> acceptAccountNo -> acceptAmount ->
              ReturnCard)

ReturnCard =  returnCard -> ((takeCard -> ATM) [] (WAIT(5);(
    retainCard -> ATM)))

Withdrawal =  amount ->
            (enoughBalance -> dispenseCash -> returnCard -> ATM)
            |~| (insufficientBalance -> returnCard -> ATM)

assert ATM :[deadlock-free]

}
```

2. `returnCard --> retainCard`
   Returning the card leads to retaining the card (expected to fail, because it holds only if the user does not take the card).

3. `withdrawCash --> dispenseCash`
   withdrawing cash leads dispensing cash (expected to fail, because the requirement holds only if there is enough cash).

4.4.2. Automated Barrier

This example illustrates a translation of an automated barrier that accepts tickets and raises the barrier exactly two time units after dispensing the ticket. The system lowers the barrier one timed unit after receiving a signal through. If the signal is not received after 20 time units of raising the barrier, it emits a beep once per time unit until the system receives a response action either *through* or *reset*. The barrier is lowered one second after the occurrence of either of these events [37].

Listing 4: specification of Automated Barrier

```
1  channel tock, acceptTicket, dispenseTicket, raiseBarrier, beep,
2         through, reset, receivedSignal, lowerBarrier
3
4  OneStep(_) = 0
5  Timed(OneStep) {
6
7  AutoBarrier = acceptTicket -> dispenseTicket -> tock -> tock ->
8              raiseBarrier -> (Response [] (WAIT(20);NoResponse))
9
10 Response = receivedSignal -> tock -> lowerBarrier -> AutoBarrier
11
12 NoResponse = Beeping /\ Action
13
14 Beeping = beep -> tock -> Beeping
15
16 Action = ((through -> SKIP) [] (reset -> SKIP))
17            ;(lowerBarrier -> AutoBarrier)
18
19 assert AutoBarrier :[deadlock-free]
20 }
```

In Listing 4, we present the specifications of Automated Barrier in *tock-CSP* that serves as input to our translation technique. Lines 1–5 are similar to the previous example. Line 7–8 describe the process `AutoBarrier` that accepts and dispenses the ticket, then waits for two times unit before raising the barrier, then behaves as a process `Timeout`. Line 10 defines the process `Timeout(P, Q, d)` that waits *d* time unit for the process `P` to perform a visible action; after the deadline *d*, the process `Timeout` behaves as *Q*. Therefore, the process `Timeout(Response, NoResponse, 20)` waits for 20 time units for the process `Response`. After the deadline elapses, the process behaves as the process `NoResponse`. Line 12 defines the process `Response` that performs the event `receivedSignal` then waits one time unit before lowering the barrier and behaving as the process `AutoBarrier`. The process `NoResponse` performs beeping infinitely (Line 16) until the process is interrupted by the process `Action`. Line 18 defines the process `Action` that presents a choice of either performing the action `through` before terminating or performing the action `reset`, that leads to lowering the barrier and behaving as the process `AutoBarrier`. Finally, Line 21 defines an assertion for checking deadlock freedom in the system. This completes the specification of `AutoBarrier`.

Verification requirements: after using our translation technique to translate *tock-CSP* into TA, the following sample requirements are specified with TCTL and verified with UPPAAL automatically.

1. ($A \diamond$ *raiseBarrier*)
   The system eventually raises the barrier.

2. ($E \diamond$ *lowerBarrier*)
   The system eventually lowers the barrier.

3. `raiseBarrier --> lowerBarrier`
   Whenever the system raises the barrier, eventually the system will lower the barrier.

4. `receivedSignal --> lowerBarrier`
   After receiving a signal, eventually the system will lower the barrier.

5. `acceptTicket --> dispenseTicket`
   After the system accepts the ticket, eventually the system will dispense the accepted ticket.

6. `beep --> lowerBarrier`
   The sound beep leads to lowering the barrier.

7. `beep` $\mathcal{U}$ `(reset` *or* `through)`
   The system continues beeping until the system receives an action either reset or through.

4.4.3. Railway Crossing System

This example illustrates a rail crossing system that consists of three components: a train, a gate, and a gate controller. The gate should be up to allow traffic to pass when no train is approaching but should be lowered to obstruct traffic when a train is close to reaching the crossing. It is the task of the controller to monitor the approach of a train and to instruct the gate to lower within the appropriate time. The train is modelled at a high level of abstraction: the only relevant aspects of the train's behaviour are when the train is near the crossing, when it is entering the crossing, when it is leaving the crossing; and the minimum delays between these events [37,126].

In *tock-CSP*, the input specification of the system is in Listing 5. The gate controller receives two types of signal from the crossing sensors: `nearInd`, which informs the controller that the train is approaching, and `outInd`, which indicates that the train has left the crossing. It sends two types of signal to the crossing gate mechanism: down command and up command, which instruct the gate to go down and up, respectively. It also receives a confirmation from the gate. These five events form the visible events of the controller. The gate, modelled by GATE, responds to the commands sent by the controller. The additional events: up and down are included to model the position of the gate [37,126].

Listing 5: Specifications of the Railway Crossing System

```
channel tock, nearInd, outInd, confirm, upCommand, downCommand,
        down, up, trainNear, enterCrossing, leaveCrossing, pass

OneStep(_) = 0

Timed(OneStep){

-- The crossing system, in conjunction with the train, is described
    as follows:
RailSystem = Train [|{nearInd, outInd}|]  Crossing

Crossing   = Controller [|{downCommand, upCommand}|] Gate

Controller = nearInd -> downCommand -> confirm -> Controller
             [] outInd -> upCommand -> confirm -> Controller

-- The gate process responds to the controller`s signals
-- by raising and lowering the gate
Gate =  downCommand  -> down -> confirm -> Gate
        [] upCommand -> up   -> confirm -> Gate

-- The process TRAIN will be used to model the approach of the train,
-- and its effect upon the crossing system

Train = trainNear -> nearInd -> enterCrossing ->
        leaveCrossing -> outInd -> pass -> Train

assert RailSystem :[deadlock-free]

}
```

Verification requirements: after using our translation technique to translate *tock-CSP* into TA, the following sample requirements are specified with TCTL and verified with UPPAAL automatically.

1. $A\diamond$ pass
   The train eventually passes the gate.

2. `trainNear --> down`
   When the train is near eventually the gate will go down

3. `leaveCrossing --> up`
   Leaving the crossing eventually leads to opening the gate.

Listing 6: Specifications of Thermostat System

```
1  channel tock, tooHot, tooCold, turnOff, turnOn
2
3  OneStep(_) = 0
4
5  Timed(OneStep){
6
7  Thermostat = (tooHot -> tock -> tock -> turnOff -> Thermostat)
8                [] (tooCold -> tock -> tock -> turnOn -> Thermostat)
9
10 assert Thermostat :[deadlock-free]
11
12 }
```

4.4.4. Thermostat System

This example illustrates a thermostat that monitors ambient temperature and controls a valve to enable or disable a heating system. The system responds to two inputs, `tooHot` and `tooCold`, which is required to perform the events `turnOff` or `turnOn` in response to these two inputs, respectively, after two time units [37].

In Listing 6, we present the *tock-CSP* specification of the system named Thermostat. It serves as input to our translation technique. Line 1–5 are similar to example 1. Line 7 defines the process `Thermostat` that presents two choices. The first choice performs the action `tooHot`, then waits two times before turning off the system, and then behaves as `Thermostat`. The second choice is the event `tooCold` that also waits two times before turning the system off and behaves as the process `Thermostat`.

Verification requirements: after translating the system into TA, the following are sample requirements that can be specified with TCTL and verified with UPPAAL automatically.

1. `tooHot --> turnOff`
   Whenever the temperature is too hot, eventually the system will turn off,

2. `tooCold --> turnOn`
   Whenever the temperature is too cold, eventually the system will turn on

3. `turnOn` $\mathcal{U}$ `tooHot`
   The system remains on until it receives a signal `tooHot`.

4. `turnOff` $\mathcal{U}$ `tooCold`
   The system remains off until it receives a signal for `tooCold`.

Listing 7: Specification of Simple Mobile system

```
1  channel obstacle, tock, moveRet, moveCall, turnCall, turnReturn
2
3  OneStep(_) = 0
4
5  Timed(OneStep) {
6
7  mSystem      = EntryMoving /\ (obstacle-> SKIP);EntryTurning;
8                               WAIT(3);mSystem
9  EntryMoving  = EDeadline(moveCall, 0);EDeadline(moveRet, 0);
10                      WAIT(1);EntryMoving
11 EntryTurning = EDeadline(turnCall, 0); EDeadline(turnReturn, 0)
12
13 EDeadline(e, t) = (e -> SKIP) [|{e}|] (WAIT(t) /\ e->SKIP)
14
15 assert mSystem :[deadlock-free]
16 }
```

4.4.5. A Simple Mobile System

This example illustrates a simple mobile system that moves at a specified linear velocity *lv* and observes the presence of an obstacle. When the system detects an obstacle, the system turns at a specified angular velocity *av* and then continues moving, repeating the procedure again [127].

In Listing 7, we present the specification of the simple mobile system. Like the previous examples, Lines 1–5 are similar to the previous test cases. Line 7 defines the process `mSystem` that starts moving until it is interrupted by an obstacle, then turns for three unit time and continues moving by behaving as the process `mSystem`. Lines 10–11 defines the process `EntryMoving` that moves for one time unit repeatedly. Line 13 defines the process `EntryTurning` for turning the system. Line 15 defines the process `Edeadline(e, t)` that specifies a deadline *t* for the event *e*. Finally, Line 17 defines an assertion for checking deadlock freedom in the specifications. This completes the specification of the simple mobile system.

Verification requirements: after using our translation technique to translate *tock-CSP* into TA, the following sample requirements are specified with TCTL and verified with Uppaal automatically.

1. `A<> obstacle`
   The system eventually detects an obstacle [14].

2. `moveCall --> obstacle`

[14]If an obstacle exists in the environment, otherwise the system will continue wondering in the operating environment.

Whenever the system moves, it will eventually detect an obstacle (if an obstacle exists in the environment).

3. `turnCall --> moveCall`
   Whenever the system turns, it will eventually moves and progress again.

4. `moveCall` $\mathcal{U}$ `obstacle`
   The system continues to move until it detects an obstacle (if an obstacle exists in the environment).

Overall, in this section, we presented sample test cases from the literature. We use a selection of six test cases considered in demonstrating the application of the translation strategy and its automated tool. Here, we provide detailed specifications of the test case as well as an overview of the specification in Table 5, which provides insight into the structure of the test cases. Also, we illustrate a selection of requirements that can be verified with TCTL, using Uppaal on the translated TA for the input *tock-CSP* specifications. As described in details in both Chapter 1 and 2, these selected requirements illustrate sample requirements that can be verified after translating the specifications.

These test cases help us in understanding both the strength and limitations of our translation technique. Also, these test cases helped us in improving and fixing errors associated with translating recursion.

4.5. Performance Evaluations

In this section, we illustrate one of the applications of our translation technique in comparing the performance of the two verification tools: FDR and Uppaal, specifically for automatic verification. We describe the resources as well as the procedure used for the performance evaluation. Previously in Section 4.3, we discussed a translation of a list of processes that enables us to compare the performance of FDR and Uppaalin this section. Evaluating the performance is another experiment that we carry out in taking advantage of the translation work.

In Section 4.3, we developed an evaluation tool that enables us to evaluate the translation technique by translating a list of formulated processes, which pairs all the considered constructors in the presented BNF of this research. We use the collection of these processes in comparing the performance of the two model-checkers: FDR and Uppaal in analysing deadlock freedom. We consider checking deadlock freedom using both FDR and Uppaal on the formulated processes constructed manually and the translated TA for Uppaal. Details of the processes are available in a provided repository of the work [117]. Each verification was repeated ten times. Details of the timings are available in the repository [117]. A summary of the average timing is presented in Figure 53, which provides an overview of comparing the performance of the tools: FDR and Uppaal.

Evaluating the performance is another experiment that we carry out for comparing the two model checkers. The graph in Figure 53 summarises the recorded timing

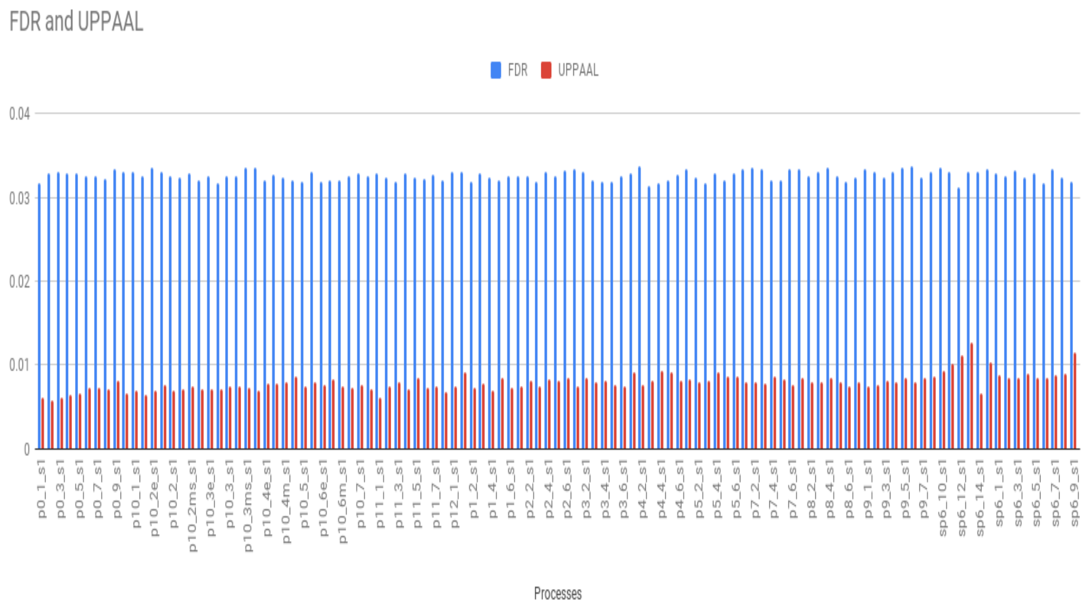


Figure 53: Performance analysis for comparing the performance of FDR and Uppaal in checking deadlock freedom (time unit in seconds).

for checking deadlock using both FDR and Uppaal, Checking deadlock is one of the common interesting checks that can be done with both FDR and Uppaal. Secondly, deadlock checks require an exhaustive verification for checking every state of each model, which makes it suitable for comparing performance.

For each process in the list of the processes we considered for the evaluation, we record the times of checking deadlock freedom using both FDR and Uppaal. Also, we record the average timing after repeating each of the checks ten times. We compare the recorded timings in Figure 53, which summarises the recorded average timing. The longer bars (blue) show the recorded timings for FDR, while shorter bars (red) show the recorded average timings for Uppaal. From the graph, it is clear that the average performance timings for FDR are above 0.03s, while the average performance timings for Uppaal is below 0.01s. This first part of the result shows that Uppaal is faster than FDR for verifying deadlock freedom.

However, with larger processes, FDR performs better than Uppaal, as shown in Table 5. The table shows the recorded average timing for each of the processes in the listed case studies (see Section 4.4). The remaining parameters provide an overview of the processes, which indicates the sizes and structures of the processes. This concludes the experiment carried out in comparing the performance of FDR and Uppaal. In the next section, we describe our plan of mathematical proof for justifying the translation rules.

4.6. An Overview of Mathematical Proofs

This section discusses our plan for using mathematical proofs in justifying the translation technique. So far, we use traces for the experimental evaluation of the translation technique. However, this is an approximation to establishing correctness with a finite set of traces. Proving correctness for the complete set of traces involves using mathematical proof. This is achieved using structural induction. An account of our initial effort to produce a proof is provided in this section. Here, we illustrate an early part of the proof with a proof of one of the base cases of the structural induction: a translation of the basic process *STOP*. In Appendix F, we provide additional details of the proof, which includes the base case *SKIP*, and the induction steps using the construct of Internal choice, External choice and Sequential composition.

For the proof of our translation function, we need to establish that, for each valid *tock-CSP* process `CSPproc`, written using the terms of the provided BNF (Section 3.1, Page 61), the following property holds.

```
   forall P::CSPproc,
       (traces_tockCSP P) = (traces_TA . transTA P)
```

Therefore, for each translation rule (Section 3.3), we need to show that the translated TA captures the behaviour of its translated construct of *tock-CSP* in the BNF (Section 3.3). In carrying out the proof we develop functions and tool that help us in developing and evaluating the proof. We use the tool in evaluating the steps of the proof to ensure that the steps are correct and consistence. The three major functions use in the proof are `trace'TA`, `traces_TA` and `traces_tockCSP`. Additional detail of these functions is available in Appendix F. The first function `traces'TA` takes a list of translated TA and computes their traces:

```
traces'TA :: [TA] -> [Traces]
```

The second function `traces_TA` takes a list of TA and returns its traces without the flow actions, define as follows:

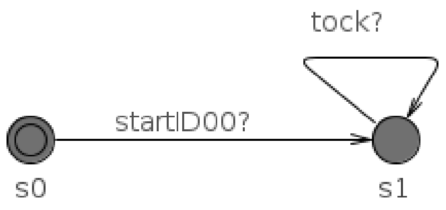
```
traces_TA :: [TA] -> [Traces]
```

Lastly, the function `traces_tockCSP` takes a *tock-CSP* process and returns its traces, as defined by Roscoe [6].

```
traces_tockCSP :: CSPProc -> [Traces]
```

For example, let us consider `TA1` as the TA for the translation of *STOP* using Rule 3.1 (Page 80). `TA1` is define as follows:

```
TA1 =  [
```



```
            ]
```

```
TA1 = [([s0, s1], s0,  [ck],
        [(startID ++ bid ++ _ ++ sid), tock],
        [(s0, (startID ++ bid ++ _ ++ sid), [], [], s1),
         (s1, tock, ck<=1, ck, s1)], [])
      ]
```

In the context of TA, a path [43,107] is a sequence of consecutive transitions that begins from the initial state, which may be an empty sequence. And a trace [43,107] (or word in the language of TA) is a sequence of actions in a given path. There is only one infinite path in TA1 with two transitions: first, a transition from location *s*0 to location *s*1; second, a transition from location *s*1 and back to the same location *s*1. The traces on the path of TA1 are expressed as follows:

```
traces'TA TA1 =
 []:[ "startID00":s |s <- [ replicate n "tock" | n <- [0..]]]
```

The function `traces'TA` computes the traces of TA1 as follows. The first empty sequence appears before the first transition. The action `startID00` happens on the first transition. The action `tock` happens on the second transition, which is repeated infinitely to produce the infinite traces on the path $\langle tock \rangle^n$.

The second function `traces_TA TA` is similar to `traces'TA TA` but removes all the coordinating actions (Definition 3.2) from the traces. In the case of `TA1` the traces are as follows:

```
tracesTA TA1 = []:[(replicate n "tock") | n <- [0..] ]
```

In the proof we use structural induction over the natural numbers.

```
forall n:Nat,
    forall P::Proc,
        traces_tockCSP n P = traces_TA n . transTA P
```

For the base case of the natural numbers ($n = 0$), we need to show that:

```
 for n=0,
    forall P::Proc,
        traces_tockCSP 0 P = traces_TA 0 (transTA P)
```

This is illustrated as follows:

```
1 proofTrans :: CSPproc -> Int -> ProofLayout [[Event]]
2 proofTrans    p          0   =
3     traces_tockCSP 0 p
4     :=: --{traces_tockCSP.0,  l~>r}
5     [[]]
6     :=: --{traces_TA.0, r~>l}
7     traces_TA 0 (transTA p  ""  0  0  0 [])
8     :=: QED
```

In ensuring that the steps of the proof are correct, we used a simplistic tool *pcpl* to support the checking of the proof. Additional detail of the tool is also included in the appendix. The tool checks the syntactic correctness and type correctness of the formulae to ensure that the steps are consistent with one another. An illustration of using the tool *pcpl* can be seen above, in the proof of the base case for integer $n = 0$.

In using the tool we describe the proof as a function that takes CSP process and an integer as universally quantified variable used in the proof layout for describing the proof. The tool `pcpl` follows the layout to check the proof. The tool has a special symbol `:=:` that separates the steps of the proof (similar to the commonly used symbol =), and Haskell comments in the form of `--{}` to indicate reasons for the steps. For example, in the above part of the proof, the steps of the proof are in Lines 3, 5 and 7, separated by the symbol `:=:`. Lines 4 and 6 are comments for the justification of the steps enclosed inside the comment symbol `--{}`. Each reason is structured to provide a justification (mostly an identifier of a function); and direction of the application of the function: right (r) to left (l) or left to right, as `l<~r` and `l~>r`, respectively; and also a mapping of the substituted value(s) which we did not use in this illustration. Line 4 describes the reasons as the application of the function `traces_tockCSP` from left to right, while Line 6 describes a reason as the application of the function `traces_TA` from right to left. For the proof of this base case ($n = 0$), since we have used no properties of `P`, other than `P::CSPproc`, we can conclude, by the rule of generalisation:

```
    forall P::Proc,
        traces_tockCSP 0 P = traces_TA 0 (transTA P)
```

4.6.1. Proof for the Construct STOP (*Base case*)

We start with the first rule (translation the construct of a basic process *STOP*; the first construct of the BNF (Section 17). For the proof by induction, we need to establish that:

```
         traces_tockCSP STOP = traces_TA (transTA STOP)
```

We need to show that

```
  (traces_tockCSP n STOP = traces_TA n (transTA STOP))
    => (traces_tockCSP (n+1) STOP = traces_TA (n+1) (transTA STOP))
```

This is illustrated as follows:

```
-- (traces_tockCSP n STOP = traces_TA n (transTA STOP))
--   => (traces_tockCSP (n+1) STOP = traces_TA (n+1) (transTA STOP))
proofTrans    STOP        n   =
    traces_tockCSP (n+1) STOP
    :=: --{traces_tockCSP.stop, n<~n+1}
    [(replicate l "tock") | l <- [0..n+1]]
    :=: --{List comprehensions }
    [(replicate l "tock") | l <- [0..n]] ++ [replicate (n+1) "tock"]
    :=: --{traces_tockCSP.stop}
    (traces_tockCSP n STOP) ++ [replicate (n+1) "tock"]
    :=: --{Induction hypothesis}
    nub ((traces_TA n (transTA STOP "ta" 0 0 0 []))
        ++ [replicate (n+1) "tock"])
    :=: --{transTA.stop, l~>r}
    nub (traces_TA n taSTOP
         ++ [replicate (n+1) "tock"])
    :=: --{tracesTA.n, l~>r, tas<~STOP}
    nub ([t \\ ["startID0_0"] |
           t <- (traces'TA n taSTOP ) ]
         ++ [replicate (n+1) "tock"] )
    :=: --{traces'TA.stop, l~>r}
    [t \\ flowActions taSTOP
            |t <- ([("startID0_0"):s
                |s <- [(replicate l "tock")|l <- [0..n] ] ] ) ]
    ++ [replicate (n+1) "tock"]
    :=: --{Introducing the connection action}
    [t \\ flowActions taSTOP
        | t <- ([("startID0_0"):s
            |s <- [(replicate l "tock")|l <- [0..n] ] ])  ]
    ++ [ t \\ ["startID0_0"]
            | t <- [["startID0_0"] ++ (replicate (n+1) "tock")] ]
    :=: --{List comprehensions}
    [t \\ flowActions taSTOP
            | t <- ([("startID0_0"):s
                |s <- [(replicate l "tock")|l <- [0..n+1] ] ] ) ]
    :=: --{traces'TA.stop, r~>l}
    nub ([t \\ flowActions taSTOP
           | t <- (traces'TA (n+1) taSTOP ) ] )
    :=: --{tracesTA.n, r~>l, n<~n+1, tas<~taSTOP}
    nub (traces_TA  (n+1)  taSTOP )
    :=: --{trans.stop, r->l}
    nub (traces_TA   (n+1)  (transTA STOP "ta" 0 0 0 []))
    :=: QED
        where
           taSTOP = [([ "s0", "s1"], "s0",  ["ck"], ["startID0_0"], [
               "tock"], [("s0", "startID0_0", [], [], "s1"), ("s1", "
               tock", "ck<=1", "ck", "s1")], [] )]
```

∴ traces_tockCSP (n+1) STOP = traces_TA (n+1) (transTA STOP)

This proves that the traces of the translated TA for *STOP* captures its traces correctly. In this section, we provide an overview of the proof with illustration of the proof for the base case. Additional details of the proof is included in the appendix.

4.7. Final Considerations

In this chapter, we described the evaluation approaches we considered in evaluating the translation technique. We discussed the implementation of the translation rules into a tool that automates the translation technique. The tool illustrates how the rules fit together as a system that automates the translation technique. This gives us confidence in using the translation rules as part of the translation strategy.

We used two types of test-cases for the evaluation. First, a collection of small processes that pair all the constructs of the *tock-CSP* within the scope of this work. The small processes have traces of sizes within the range 0–10, except recursive processes that have unbounded traces. The second collection of test cases is processes of reasonable size systems for evaluating the translation technique. However, their traces are infinite due to the nature of the timed system that must allow the progress of time.

We performed the experiment on Ubuntu 16.04.6 LTS (Xenial Xerus) running on Intel Core i7-4700MQ (4C/8T, 2.4GHz) processor with 32GB RAM (4x 8GB modules). This gives us a good opportunity for comparing the performance of FDR and UPPAAL, in analysing deadlock freedom. It would be interesting to see a general comparison of the tools, but that is a topic for future work. In this work, we take an average of ten running for each process. The result shows that UPPAAL performs better on small processes. However, with more substantial processes, FDR performs better than UPPAAL in comparing the performance experiment on the larger test-cases, as described in this chapter.

Additionally, we have described an initial mathematical proof for establishing the correctness of the translation rules, given that mathematical proofs provide greater certainty than trace analysis. Thus, we plan to use structural induction to prove the translation rules. The mathematical proof will give higher assurance for justifying the translation rules.

These two validation considerations serve to validate the correctness of the translation technique, and shows that our strategy is a promising basis for a translating *tock-CSP* into TA. In the next chapter, we evaluate the presented work, present conclusion, and discuss future work.

Chapter Five

5. Summary and Conclusion

We conclude by summarising the research work conducted in this study, in Section 5.1. Then, in Section 5.2, we discuss the limitations of the work and recommend future directions for improving the work.

5.1. Summary

In this research work, we have discussed the available resources for the verification of robotics applications in Chapter 2. Perhaps the most fundamental question that emerged from early exploratory work was how to improve formal techniques of verifying temporal specifications that are compatible with the existing resources of developing robotics, especially the ones that have a formal basis (Chapter 1). In investigating this question, we studied the available formal techniques for verifying temporal specifications. We found that various techniques and tools have been developed for supporting the advancement of robotics software. However, there is less attention given to verifying the correctness of the robotics applications; for instance, few techniques have been developed to support the verification of the robotics software.

In the literature (Chapter 2), we studied the existing technique and tools available in the area of software engineering that focus on three directions: Formal Method, DSML and Temporal specifications. We found that many DSMLs have been developed to improve the advancement of robotics software systems. However, most of the techniques lack a formal basis, and there is little proper support for verification, specifically in verifying temporal specifications (Chapter 1 and 2).

On the basis of four criteria: modelling, reasoning, system and tool support, we studied a selection of nine DSMLs where we found many interesting resources and features for enhancing the development of robotics systems (Chapter 2). Based on the research conducted for this work, there is less attention given to enhancing the verification aspect, especially in verifying temporal specifications with respect to real-time. This indicates the need for improving the verification aspect specifically in verifying temporal specification.

In the area of formal method, *tock-CSP* has been developed for modelling temporal specifications with the support of FDR for automatic verification. Among the studied DSMLs, RoboChart uses *tock-CSP* for verifying temporal specifications. Also, we find that there are existing projects and tools that provide facilities for verifying temporal specification in a continuous-time model, particularly in the formal verification of real-time systems. For instance, UPPAAL, KRONOS and PRISM are popular real-time verification tools. From the literature, UPPAAL is the most advanced and most efficient tool that supports the verification of TA with TCTL (Chapter 2).

In Chapter 3, we presented a proposed technique for translating *tock-CSP* into TA for UPPAAL, which facilitates using temporal logic, and automatic support with UPPAAL in verifying *tock-CSP* models. This translation technique provides a way of using TCTL in specifying liveness requirements that are difficult to specify and verify in *tock-CSP*. Also, the result of this work sheds additional insight into the complex relationship between *tock-CSP* and TA, as well as the connection between the refinement model and temporal logic model.

We extended *tock-CSP* with the additional support of real-time model-checker UPPAAL for verifying temporal specifications, especially for the kind of requirements that are difficult to verify with FDR, such as liveness specifications. We developed a translation technique for translating *tock-CSP* into TA that facilitates using UPPAAL for automatic verification of *tock-CSP*, which improves the facilities of verifying temporal specification in RoboChart and other existing works that are based on *tock-CSP*.

In developing the translation technique, we provided a BNF that serves as a link for connecting existing works in *tock-CSP* with our translation technique. We developed a translation strategy for the translation technique. Based on the strategy, we came up with translation rules that describe the translation of each construct of the *tock-CSP* into TA, within the scope of our work (Chapter 3).

We considered translating *tock-CSP* models into a network of small TAs that meets the specifications of *tock-CSP*. This is due to the compositional characteristics of *tock-CSP* (inherited from CSP) which is not available in TA, and it is challenging translating *tock-CSP* into a single TA. With our approach of using a network of small TAs, we captured the specifications of the *tock-CSP* into TA, which facilitates using TCTL for the verification of *tock-CSP* models.

In justifying the correctness of the translation strategy (Chapter 4), we used trace analysis to compare the behaviour of the input *tock-CSP* with the behaviour of the translated TA. We generated the traces in two stages. In stage 1, we generated traces from both FDR and UPPAAL; if the traces did not match, we created stage 2 of the trace analysis. In stage 2, we used UPPAAL to verify that all the traces of *tock-CSP* are valid traces of the translated TA. In this case, we used the power of FDR to complement the power of UPPAAL in generating traces. This is because UPPAAL is not capable of generating traces that evaluate to the same logical value irrespective of the permutation of the events.

In evaluating the translation technique, we used two categories of test cases. First, we used a collection of formulated processes for pairing all the essential *tock-CSP* constructs, within the scope of this translation work. Second, a selection of test cases from the literature were used that helped us to understand the strengths and limitations of the translation work. The selection of the test cases is in the wider sense as defined from the IEEE Standard 1872-2015. The collection of these processes gives us the opportunity to compare the two model checkers: FDR and UPPAAL in analysing deadlock freedom where the input *tock-CSP* process is constructed manually and the translated TA generated by our translation tool. Because of that, the results may not be a fair comparison, as the generated TA may not be the most efficient models for UPPAAL. The result shows that FDR performs better in the larger processes, while UPPAAL performs better in the smaller processes (Chapter 4). However, the boundary between the range of the processes remains unknown; an interesting investigation for future work.

Considering that trace analysis is limited to bounded traces, which is limited to covering safety properties, it is clear that trace analysis covers a partial notion of correctness that will not be able to cover infinite traces. Thus, we planned to extend the trace analysis by using mathematical proofs to establish the correctness of the translation techniques that will cover infinite traces. This was achieved with structural induction. We provided an illustration of the proof for the base cases and an induction step. Completing the proof will provide an additional convincing justification for the technique and its supporting tool. This concludes our contribution for this research work.

5.2. Future work

In this section, we discuss the limitations that are beyond the scope of our work. Also, we provide insight into further work that can improve this work in a promising direction of the future work.

So far, in this work, we have used trace analysis to justify the correctness of the translation work. We provided a good plan for using structural induction to prove the correctness of the translation rules. Completing the proof is beyond the scope of this work. An immediate recommended future work is to complete the proof for the correctness of the translation technique. Besides, the scope of this work is limited by trace analysis. It will be interesting if future work will consider expanding the scope of the proof to include refusal and divergences.

Unifying Theories of Programming (UTP) provides a unified framework for comparing semantics of programs that facilitates reasoning about programming theories as well as links between programmes and theories. To increase the strength of the translation work, it will be interesting to see an expansion of this work by establishing the correctness of the translation technique via UTP. For instance, by building a semantic link between *tock-CSP* and TA using operational semantics of both sides.

The provided BNF (Chapter 3) covers the essential operators of *tock-CSP*. A natural extension of this work is to expand the BNF to cover all the operators of *tock-CSP*. This will improve the application of the translation technique to translate a broader category of systems.

In this work, we considered a network of small TAs to develop the translation technique due to the problem of handling the compositional structure of *tock-CSP*, as mentioned in Chapter 3. It will be interesting to explore translating the *tock-CSP* into a large size TA, instead of the small sizes. This has the potential of improving the performance of UPPAAL for automatic verification of the translated TA models.

Also, using the translation technique to translate extensive case studies will help to further understand the strengths and robustness of our translation work, as well as additional scope that will limit the application of the translation technique. For instance, in translating the case studies (Section 4.4), we understand the role of translating data which makes it necessary to omit additional interesting case studies that we may like to include, such as Triple Redundancy Protocol, Grid system and Conway puzzle. Expanding the work to include translating data will be a good addition to this work.

Additionally, currently, we translated the event `tock` into an action that is controlled by a timed clock in UPPAAL. A recommended next step in this work is to relate the notion of `tock` to the notion of time in TA and getting rid of *tock* as an action; such that we can be able to translate a process like `P=tock->tock->tock-SKIP` (or `WAIT(3)`) into a single TA, like in Figure 54, which has clock `ck>3` that captures the time. This additional extension will help us to explore additional interesting facilities of UPPAAL for verifying temporal specifications.

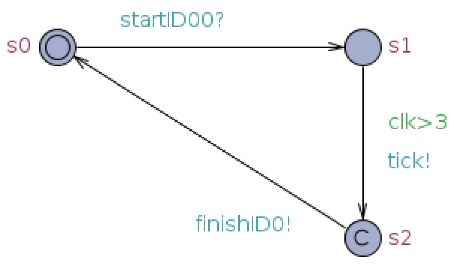


Figure 54: A sketch of a sample translated TA that illustrates a translation of the process $WAIT(3)$ with clock $clk > 3$.

Adding a GUI to the translation tool will be a significant improvement to the tool. For instance, it will be interesting to see users interacting with the tool through the GUI. Additionally, incorporating feedback from the results of the analysis directly to the GUI will make it easier to understand the analysis.

This translation work opens up other directions for future work. For instance, translating the refinement specifications into TCTL will be a good addition to this work. This will make it easy to communicate results between FDR and UPPAAL.

Another interesting extension of this work is calculating the complexity of the translation technique and the generated networks of timed automata. This will be useful for further work in improving the efficiency of both the translation technique and the translated TA. Although this may not be simple work because many factors need to be considered in describing the complexity, apart from the size of the input process, other factors include a compositional structure of the input process, a combination of operators used in describing the input process, the structure of the input process (such as recursive, deadlock, and termination) and also both concurrency and the number of synchronisation points in the process.

Another interesting work is to expand the performance experiment (Section 4.5) to find out additional details for answering the pending question of why UPPAAL performs better in smaller processes and FDR performs better in larger processes. It will be interesting to know the clear boundary between the performance of the two tools in the future extension of this work.

It would be interesting to see how our translation work will match the evolution of RoboChart models. At the time conducting this work, RoboChart was still evolving. As such, we focus on *tock-CSP* that captures the generated semantics of the RoboChart models. Part of the future work should investigate the application of this work when a stable version of RoboChart become available.

In addition, it will be interesting to include other tools such that we can understand the strengths of these tools and utilise them appropriately. There are still many other aspects that can be further explored in this research direction. These include consideration of translating *tock-CSP* into KRONOS, which also provides a verification engine for verifying TA in modelling and verification of system specifications. It has a different approach from UPPAAL. This completes the work we carried out in this research.

Appendices

A. Complete Translation Rule of Prefix

This section describes the translation of operator `Prefix`. The section begins with presenting a rule for translating the operator `Prefix`, and then follows with an example that illustrates using the rule in translating a process.

This rule for translating prefix happens to be the largest translation rule because we can not translate an event without knowing whether the event used in the prefix is, in the overall process, being hidden, renamed, or used as part of synchronisation, initial of external choice or initial of interrupt. In each case the event has different behaviour.

Thus, first the translation rule includes a function for checking both hidden and renamed events. And then we formulate eight cases for capturing the possible behaviour of an event that is used as part of synchronisation, external choice or interrupt. Each case captures a possible behaviour of an event for a process that is part of synchronisation, external choice or interrupt.

Rule A.1. Translation of Prefix (1 of 5))

```
transTA (Prefix e1 p) procName bid sid fid usedNames =
        (([(TA idTA [] [] locs1 [] (Init loc1) trans1)] ++ ta1),
                sync1, syncMapUpdate)
  where
    idTA = "taPrefix" ++ bid ++ show sid
    (syncs, syncMaps, hides, renames, exChs, intrpts,
       initIntrpts,
          excps) = usedNames

    -- Checking hiding or renaming
    e = checkHidingAndRenaming e1 hides renames

    {- High level definition of locations and transitions for
       the eight possible combination of synchronisation, choice
        and interrupt, 000, 001, 010, 011, 100, 101, 110, 111 -}
    (locs1, trans1)
        |((not synch)&&(not exChoice)&&(not interupt)) = case1
        |((not synch)&&(not exChoice)&&(    interupt)) = case2
        |((not synch)&&(    exChoice)&&(not interupt)) = case3
        |((not synch)&&(    exChoice)&&(    interupt)) = case4
        |((    synch)&&(not exChoice)&&(not interupt)) = case5
        |((    synch)&&(not exChoice)&&(    interupt)) = case6
        |((    synch)&&(    exChoice)&&(not interupt)) = case7
        |((    synch)&&(    exChoice)&&(    interupt)) = case8
```

Translation of Prefix (2 of 5)

```
    case1 = ([loc1, loc2, loc5],
             [t12, t25,  t51] ++ addTran ++ transIntrpt')
    case2 = ([loc1, loc2, loc3c, loc5],
             if not $ null intrpts
             then [t12G, t23ci, t3c5, t51] ++ addTran
             else [t12,  t23ci, t23cgi, t3c5, t51] ++ addTran
                  ++ transIntrpt')
    {- if a process can interrupt and also be interrupted, then
       it can only be interrupted after initiating its interrupt
        -}
    case3 = ([loc1, loc2,  loc3c,  loc5],
             [t12,  t23c,  t3c5,    t51] ++ t23e ++ addTran
             ++ transIntrpt')
    case4 = ([loc1, loc2,  loc3c,  loc4c,   loc5],
             [t12G, t23c,  t3c4ci, t3c4cgi, t4c5e,  t51] ++
             addTran ++ transIntrpt')
    case5 = ([loc1, loc2, loc3, loc5],
             [t12,  t23,  t35,  t51] ++ addTran ++ transIntrpt')
    case6 = ([loc1, loc2,  loc3,   loc4c,   loc5],
             [t12G, t23,   t34c,   t4c5i,   t4c5gi, t51]  ++
             addTran ++ transIntrpt')
    case7 = ([loc1, loc2,  loc3,   loc4,    loc5],
             [t12,  t23ech, t34,   t45,     t51] ++
             addTran ++ transIntrpt')
    case8 = ([loc1, loc2,  loc3,   loc4,    loc5,   loc6],
             [t12G, t23,   t34c,   t4c6,    t65, t65gi, t51] ++
             addTran ++ transIntrpt')

    --      =  Location  ID    Name  Label       LocType
    loc1    =  Location "id1"  "s1"  EmptyLabel None
    loc2    =  Location "id2"  "s2"  EmptyLabel None
    loc2c   =  Location "id2"  "s2"  EmptyLabel CommittedLoc
    loc3    =  Location "id3"  "s3"  EmptyLabel None
    loc3c   =  Location "id3"  "s3"  EmptyLabel CommittedLoc
    loc4    =  Location "id4"  "s4"  EmptyLabel None
    loc4c   =  Location "id4"  "s4"  EmptyLabel CommittedLoc
    loc5    =  Location "id5"  "s5"  EmptyLabel CommittedLoc
    loc6    =  Location "id6"  "s6"  EmptyLabel CommittedLoc

    transIntrpt' = (transIntrpt intrpts loc1 loc2)

    -- Additional transitions for tock, external choice
    addTran | ((not  $ elem e syncs)&&(null exChs))      = [t22]
            | ((elem e syncs)       &&(null exChs))      = [t22]
            | ((not  $ elem e syncs)&&(not $ null exChs))= [t22]
                 ++ t21
            | otherwise = [t22, t33, t44] ++ t21
```

Translation of Prefix (3 of 5)

```
    t23ci   = Transition  loc2     loc3c   l23ci            []
    t23cgi  = Transition  loc2     loc3c   altIntrpt        []
    l23ci   = [(Sync (VariableID   ((show e) ++ "_intrpt")  [])
       Excl), (Update  [(AssgExp  (ExpID ((show e) ++ "
       _intrpt_guard")) ASSIGNMENT TrueExp )])]

    {- An alternative transition in case another event has
       already initiates the interrupt. Guard other possible
       interrupts, such that any of the interrupt can enable the
        alternative transition -}
    altIntrpt = [(Guard
                  (ExpID (intercalate "␣||␣"  [l ++
                  "_intrpt_guard"| (ID l) <- initIntrpts])))]
    -- reset the guards in case of recursive process
    resetG  = [Update  [(AssgExp  (ExpID (l ++ "_intrpt_guard"))
                    ASSIGNMENT FalseExp)| (ID l) <- initIntrpts
                        ]]
    t3c5    = Transition  loc3c    loc5    lab4e            []
    t3c4ci  = Transition  loc3c    loc4c   l23ci            []
    t3c4cgi = Transition  loc3c    loc4c   altIntrpt        []
    t4c5    = Transition  loc4c    loc5   (lab2i ++ lab2d)  []
    t4c5e   = Transition  loc4c    loc5    lab4e            []
    t34c    = Transition  loc3     loc4c   lab4             []
    t3c4c   = Transition  loc3c    loc4c   lab4e            []
    t4c5i   = Transition  loc4c    loc5    l23ci            []
    t4c5gi  = Transition  loc4c    loc5    altIntrpt        []
    t4c6    = Transition  loc4c    loc6   (lab2i ++ lab2d)  []
    t65     = Transition  loc6     loc5    l23ci            []
    t65gi   = Transition  loc6     loc5    altIntrpt        []
    t12     = Transition  loc1     loc2    lab1             []
    t12G    = Transition  loc1     loc2   (lab1 ++ resetG)  []
    t23     = Transition  loc2     loc3    lab2i            []
    t2c3    = Transition  loc2c    loc3    lab2i            []
    t23c    = Transition  loc2     loc3c  (lab2i ++ lab2d)  []
    t23ech  = Transition  loc2     loc3   (lab2i ++ lab2d)  []
    t25     = if elem e hides
              then Transition    loc2  loc5
                ([(Sync (VariableID "itau" []) Excl)]) []
              else Transition    loc2  loc5   lab2i    []
    t25r    =  Transition  loc2  loc5  ([(Sync (VariableID
                    (show new_e) []) Excl)] ++ labpath) []
    new_e   =  head [newname | (oldname, newname) <- renames,
       oldname == e]
    t51     =  Transition  loc5  loc1  [lab3]          []
    t33     =  Transition  loc3  loc3  [labTock]       []
    t44     =  Transition  loc4  loc4  [labTock]       []
    t35     =  Transition  loc3  loc5   lab4           []
    t22     =  Transition  loc2  loc2  [labTock]       []
```

Translation of Prefix (4 of 5)

```
    t21      = [(Transition loc2  loc1  [(Sync  (VariableID
                  ((show ch) ++ "_exch")  []) Ques)] [])|ch <-
                      exChs']
    t23e     = [(Transition loc2  loc3  [(Guard (ExpID ((show ch)
          ++ "_exch_ready")) )] [])|ch <- exChs']
    t34      =  Transition  loc3  loc4   lab6  []
    t45      =  Transition  loc4  loc5   lab4  []

    lab1     = [Sync (VariableID (startEvent procName bid sid)
         []) Ques]

    lab2i    | (elem e syncs)&&(null exChs')  =    -- check sync
                  [(Guard (BinaryExp (ExpID ("g_" ++
                       (eTag e syncMaps' ""))) Equal (Val 0))),
                  (Update [(AssgExp (ExpID ("g_" ++
                       (eTag e syncMaps' ""))) AddAssg (Val 1))])]
             | (not $ null exChs') =
                       if (elem e hides)
                       then [(Sync (VariableID "itau_exch"  [])
                           Excl)]
                       else [(Sync (VariableID ((show e ) ++
                         "_exch")     []) Excl)]
             | otherwise    =  lab4e

    labpath = [(Update  [(AssgExp (ExpID "dp") AddAssg (Val 1)),
               (AssgExp  ( ExpID   ("ep_" ++ bid ++ "_" ++ show
                   sid)) ASSIGNMENT TrueExp )])]
                  -- Attaching path variable transition
    -- Checks for exception
    lab3     =  if elem e (fst excps)
                then Sync (VariableID ("startExcp" ++ (show fid )
                    ) []) Excl
                else Sync (VariableID ("startID" ++ bid ++ "_" ++
                           show (sid+1)) []) Excl
    lab4     = [(Sync (VariableID ((show e) ++ "___sync") [])
        Ques)]
    lab4e    | e == Tock    = [(Sync (VariableID (show e) [])
        Ques)] ++ labpath  -- Sync on tocks
             | elem e hides = [(Sync (VariableID ("itau") [])
                Excl)]     -- itau for hiding event
             | otherwise    = [(Sync (VariableID (show e) [])
                Excl)]  ++ labpath  -- Fire normal event
    lab6     = [(Guard (BinaryExp (ExpID ("g_" ++ (eTag e
        syncMaps' ""))) Equal (Val 0))), (Update [(AssgExp (ExpID
         ("g_" ++ (eTag e syncMaps' ""))) AddAssg (Val 1))])]
```

Translation of Prefix (5 of 5)

```
    lab2d   = [(Update [(AssgExp  (ExpID ((show ch) ++ "_exch_ready")) AddAssg  (Val 1)) | ch <- exChs'])]
    gIntrpt = [(Guard (BinaryExp (ExpID "gIntrpt") Equal   (Val 1)))]
    uIntrpt = [(Update [(AssgExp (ExpID "gIntrpt") AddAssg (Val 1))])]
    labTock = Sync (VariableID  "tock" [])  Ques
    synch    = elem e syncs
    exChoice = null exChs
    interupt = null initIntrpts

    -- Update sync points
    syncMaps' = if elem e syncs
                then [(e, (show e) ++ bid ++ "_" ++ show sid )]
                else [] -- syncMaps_

    -- Combine the synchronisations together
    syncMapUpdate = syncMaps' ++ syncMap1

    -- Replace renamed event with the new name
    exChs' =  if  ( null crs ) then exChs
              else  (exChs \\ [es']) ++ [nn']

    -- rename all events for blocking external choice
    crs  = [(es, nn) | (es, nn) <- renames, ch <- exChs, ch == es]
    (es', nn') = head crs

    {- Update used names and then remove external choice and interrupt if any, after the first event. -}
    usedNames' = (syncs, syncMaps, hides, renames, [], intrpts, [], excps)

    -- Finally recursive call for subsequent translation.
    (ta1, sync1, syncMap1) = transTA p []  bid (sid+1) fid usedNames'
```

B. List of Small Processes for the Experimental Evaluation

This section describes an overview of the list of processes we use for the experimental evaluation. Table 6 describes the constant processes, while Table 7 describes a list for processes for pairing the constructs we consider in this work. The processes in Table 7 illustrates an exhaustive pairing of the constructs with one another.

01	STOP	Deadlock (Dl)
02	Stopu	Urgent Deadlock (UD)
03	SKIP	Termination (Tn)
04	Skipu	Urgent Termination (UT)
05	WAIT(n)	Delay (Dy)
06	Waitu(n)	Strict Delay (SD)

Table 6: Basic processes for the experimental evaluation

BNF Constructors	Dl	UD	Tn	UT	Dy	SD	Px	IC	EC	Il	GP	SC	RP	To	It	ED	Hd	Rn	Ex
Prefix (Px)	10	11	12	13	14	15	16	17	18	19	1A	1B	1C	1D	1E	1F	1G	1H	1J
Internal Choice (IC)	20	21	22	23	24	25	26	27	28	29	2A	2B	2C	2D	2E	2F	2G	2H	2J
External Choice (EC)	30	31	32	33	34	35	36	37	38	39	3A	3B	3C	3D	3E	3F	3G	3H	3J
Interleaving (Il)	40	41	42	43	44	45	46	47	48	49	4A	4B	4C	4D	4E	4F	4G	4H	4J
Generallise Parallel (GP)	50	51	52	53	54	55	56	57	58	59	5A	5B	5C	5D	5E	5F	5G	5H	5J
Sequential (SC)	60	61	62	63	64	65	66	67	68	69	6A	6B	6C	6D	6E	6F	6G	6H	6J
Recursive Process (RP)	70	71	72	73	74	75	76	77	78	79	7A	7B	7C	7D	7E	7F	7G	7H	7J
Interrupt (It)	80	81	82	83	84	85	86	87	88	89	8A	8B	8C	8D	8E	8F	8G	8H	8J
Timeout (To)	90	91	92	93	94	95	96	97	98	99	9A	9B	9C	9D	9E	9F	9G	9H	9J
Event Deadline (ED)	A0	A1	A2	A3	A4	A5	A6	A7	A8	A9	AA	AB	AC	AD	AE	AF	AG	AH	AJ
Hiding (Hd)	B0	B1	B2	B3	B4	B5	B6	B7	B8	B9	BA	BB	BC	BD	BE	BF	BG	BH	BJ
Renaming (Rn)	C0	C1	C2	C3	C4	C5	C6	C7	C8	C9	CA	CB	CC	CD	CE	CF	CG	CH	CJ
Exception (Ex)	D0	D1	D2	D3	D4	D5	D6	D7	D8	D9	DA	DB	DC	DD	DE	DF	DG	DH	DJ

Table 7: An overview of the small processes for pairing the constructors used in the experimental evaluation.

C. Abstract Syntax Tree of UPPAAL TA

Source [15]

```
BNF for the 4.x XTA format
            XTA ::= <Declaration>* <Instantiation>* <System>
    Declaration ::= <FunctionDecl> | <VariableDecl> | <TypeDecl> | <
        ProcDecl>
  Instantiation ::= ID ASSIGNMENT ID '(' <ArgList> ')' ';'
         System ::= 'system' ID (',' ID)* ';'

  ParameterList ::= '(' [ <Parameter> ( ',' <Parameter> )* ] ')'
      Parameter ::= <Type> [ '&' ] ID <ArrayDecl>*

   FunctionDecl ::= <Type> ID <ParameterList> <Block>

       ProcDecl ::= 'process' ID <ParameterList> '{' <ProcBody> '}'
       ProcBody ::= (<FunctionDecl> | <VariableDecl> | <TypeDecl>)*
                    <States> [<Commit>] [<Urgent>] <Init> [<
                        Transitions>]

         States ::= 'state' <StateDecl> (',' <StateDecl>)* ';'
      StateDecl ::= ID [ '{' <Expression> '}' ]

         Commit ::= 'commit' StateList ';'
         Urgent ::= 'urgent' StateList ';'
      StateList ::= ID (',' ID)*

           Init ::= 'init' ID ';'

    Transitions ::= 'trans' <Transition> (',' <TransitionOpt>)* ';'
     Transition ::= ID '->' ID <TransitionBody>
  TransitionOpt ::= Transition | '->' ID <TransitionBody>
 TransitionBody ::= '{' [<Guard>] [<Sync>] [<Assign>] '}'

          Guard ::= 'guard' <Expression> ';'
           Sync ::= 'sync' <Expression> ('!' | '?') ';'
         Assign ::= 'assign' <ExprList> ';'

       TypeDecl ::= 'typedef' <Type> <TypeIdList> (',' <TypeIdList>)*
           ';'
     TypeIdList ::= ID <ArrayDecl>*
BNF for variable declarations
   VariableDecl ::= <Type> <DeclId> (',' <DeclId>)* ';'
         DeclId ::= ID <ArrayDecl>* [ ASSIGNMENT <Initialiser> ]
    Initialiser ::= <Expression>
                 |  '{' <FieldInit> ( ',' <FieldInit> )* '}'
```

[15] Taken from UPPAAL documentation in this link http://people.cs.aau.dk/~adavid/utap/syntax.html

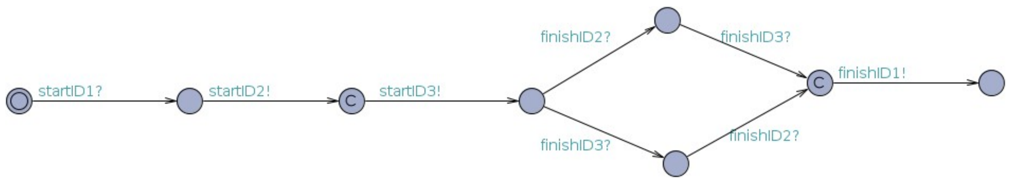
```
      FieldInit ::= [ ID ':' ] <Initialiser>

      ArrayDecl ::= '[' <Expression> ']'

           Type ::= <Prefix> ID [ <Range> ]
                 |  <Prefix> 'struct' '{' <FieldDecl>+ '}'
      FieldDecl ::= <Type> <FieldDeclId> (',' <FieldDeclId>)* ';'
    FieldDeclId ::= ID <ArrayDecl>*

         Prefix ::= ( [ 'urgent' ] [ 'broadcast' ] | ['const'] )
          Range ::= '[' <Expression> ',' <Expression> ']'


-- BNF for statements
    Block ::= '{' ( <VariableDecl> | <TypeDecl> )* <Statement>* '}'
Statement ::= <Block>
           |  ';'
           |  <Expression> ';'
           |  'for' '(' <ExprList> ';' <ExprList> ';'
                   <ExprList> ')' <Statement>
           |  'while' '(' <ExprList> ')' <Statement>
           |  'do' <Statement> 'while' '(' <ExprList> ')' ';'
           |  'if' '(' <ExprList> ')' <Statement>
                   [ 'else' <Statement> ]
           |  'break' ';'
           |  'continue' ';'
           |  'switch' '(' <ExprList> ')' '{' <Case>+ '}'
           |  'return' ';'
           |  'return' <Expression> ';'

Case      ::= 'case' <Expression> ':' <Statement>*
           |  'default' ':' <Statement>*

-- BNF for expressions
   ExprList ::= <Expression> ( ',' <Expression> )*
 Expression ::= ID
            |   NAT
            |   'true' | 'false'
            |   ID '(' <ArgList> ')'
            |   <Expression> '[' <Expression> ']'
            |   '(' <Expression> ')'
            |   <Expression> '++' | '++' <Expression>
            |   <Expression> '--' | '--' <Expression>
            |   <Expression> <AssignOp> <Expression>
            |   <UnaryOp> <Expression>
            |   <Expression> <Rel> <Expression>
            |   <Expression> <BinIntOp> <Expression>
            |   <Expression> <BinBoolOp> <Expression>
            |   <Expression> '?' <Expression> ':' <Expression>
            |    <Expression> '.' ID>

   AssignOp ::= ASSIGNMENT | '+=' | '-=' | '*=' | '/=' | '%='
            | '|=' | '&=' | '^=' | '<<=' | '>>='
    UnaryOp ::= '-' | '!'
        Rel ::= '<' | '<=' | '==' | '!=' | '>=' | '>'
   BinIntOp ::= '+' | '-' | '*' | '/' | '%' | '&' | '|' | '^' | '<<'
       | '>>'
  BinBoolOp ::= '&&' | '||'
    ArgList ::= [ <Expression> ( ',' <Expression> )* ]
```

D. Additional Examples

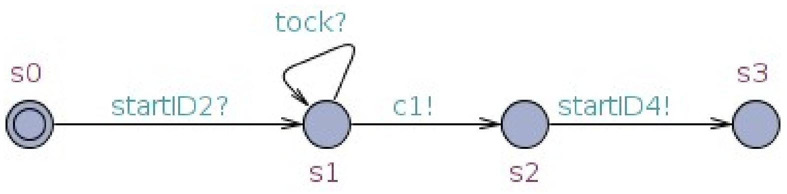
Additional examples are provided here to illustrate possible synchronisation patterns of the generallised parallel operator. The process P1, Q1 and R1 can be arbitrary processes. But for the purpose of illustration, we consider simple definition for each of these processes P1, Q1 and R1 as follows:
P1 = `c1->cs->SKIP`
Q1 = `c2->cs->SKIP`
R1 = `c3->cs->SKIP`

Example D.1. $P1[|\{cs\}|](Q1[|\emptyset|]R1)$

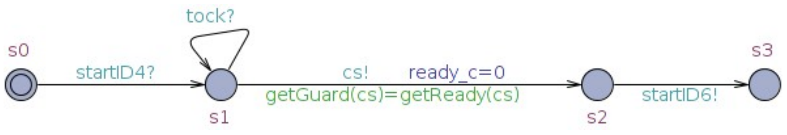
```
transform(P[|{cs}|](Q1[|∅|]R1)))
 = transTA(P1[|{cs}|](Q1[|∅|]R1)), startID1, finishID1, ∅, ∅, ∅,)
 = [
```



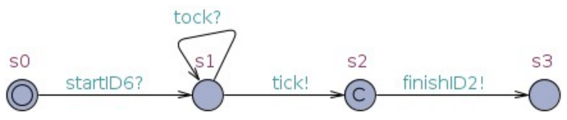
```
        ] ⌢ transTA(P1, startID2, finishID2, ∅, {cs} ∅)
        ⌢ transTA((Q1[|∅|]R1), startID3, finishID3, {cs}, ∅, ∅)

transTA(P1, startID2, finishID2, ∅, {cs} ∅)
 = transTA(c1->cs->SKIP, startID2, finishID2, ∅, {cs} ∅)
 = [
```



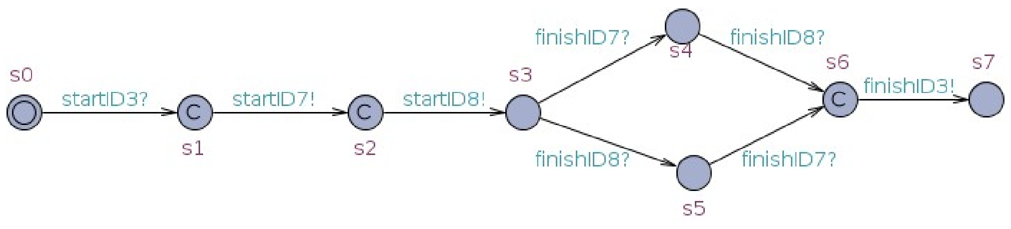
```
    ] ⌢ transTA(cs->SKIP, startID4, finishID2, ∅, {cs} ∅)
       = [
```



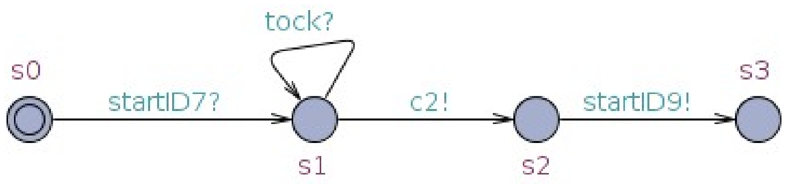
```
      ] ⌢ transTA(SKIP, startID6, finishID2, ∅, {cs} ∅)
          = [
```



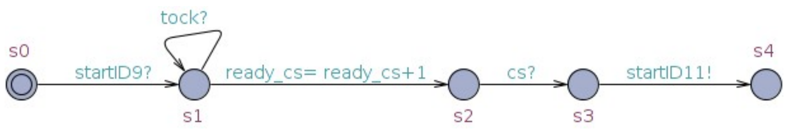
```
]

= transTA((Q1[|∅|]R1), startID3, finishID3, {cs}, ∅, ∅)
= [
```



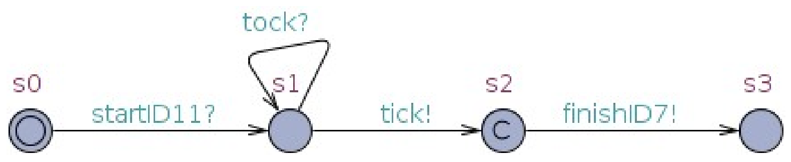
```
        ] ⌢ transTA(Q1, startID7, finishID7, {cs}, ∅, ∅)
        ⌢ transTA(R1, startID8, finishID8, {cs}, ∅, ∅)

transTA(Q1, startID7, finishID7,  {cs}, ∅, ∅)
 = transTA(c2->cs->SKIP, startID7, finishID7,  {cs}, ∅, ∅)
 =[
```



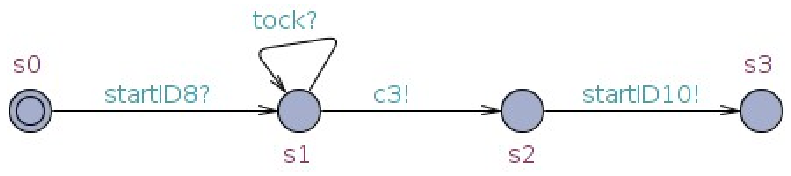
```
      ] ⌢ transTA(cs->SKIP, startID9, finishID7, {cs}, ∅, ∅)
```



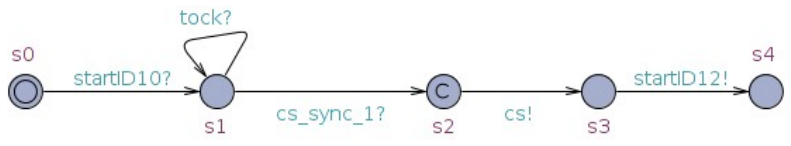
```
      ] ⌢ transTA(SKIP, startID11, finishID7, {cs}, ∅, ∅)
      = [
```



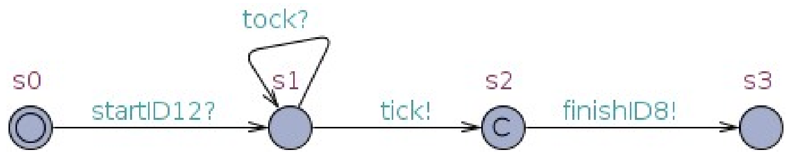
```
]

transTA(R1, startID8, finishID8,  {cs}, ∅, ∅)
 = transTA(c3->cs->SKIP, startID8, finishID8,  {cs}, ∅, ∅)
 = [
```



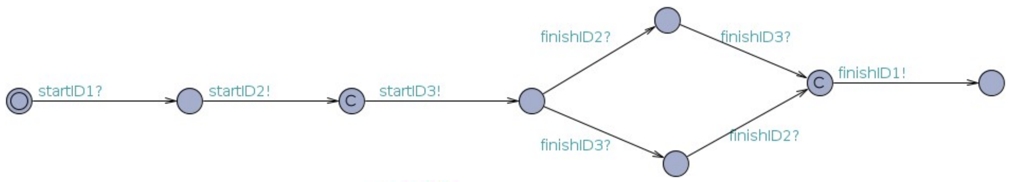
```
    ] ⌢ transTA(cs->SKIP, startID10, finishID8, {cs}, ∅, ∅)
    = [
```



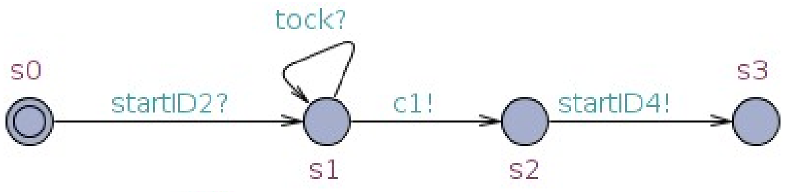
```
    ] ⌢ transTA(SKIP, startID8, finishID8, {cs}, ∅, ∅)
    = [
```



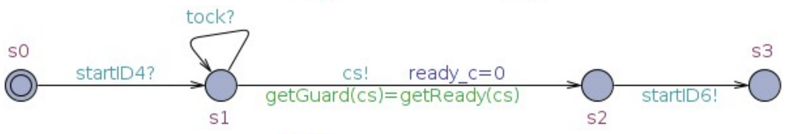
```
]

∴ transform(P1[|{cs}|](Q1[|∅|]R1))
 = [
```



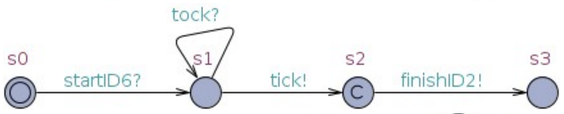
startID1?
startID2!
C
startID3!
finishID2?
finishID3?
finishID3?
finishID2?
C
finishID1!



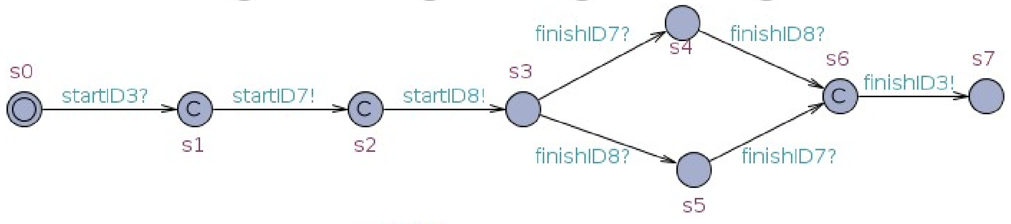
tock?
s0
startID2?
c1!
startID4!
s3
s1
s2



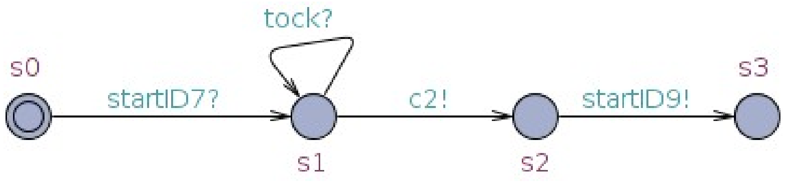
tock?
s0
startID4?
cs!
ready_c=0
getGuard(cs)=getReady(cs)
s3
startID6!
s1
s2



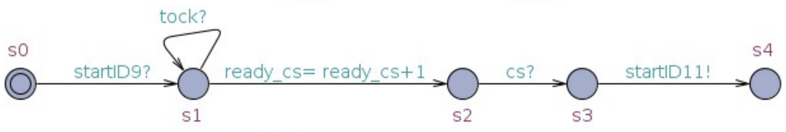
tock?
s0
startID6?
s1
tick!
s2
C
finishID2!
s3



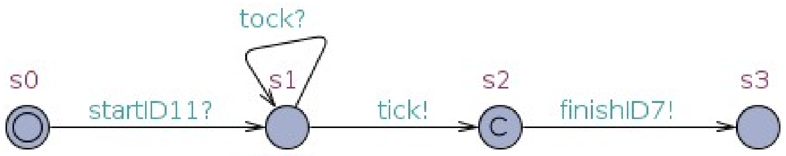
s0
startID3?
C
s1
startID7!
C
s2
startID8!
s3
finishID7?
s4
finishID8?
finishID8?
s5
finishID7?
s6
C
finishID3!
s7



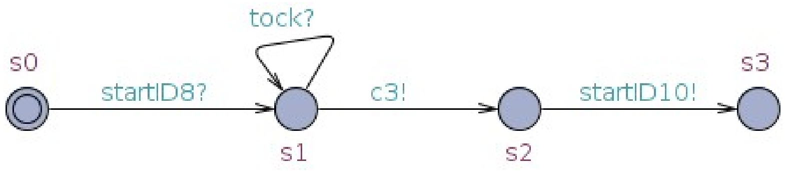
tock?
s0
startID7?
c2!
startID9!
s3
s1
s2



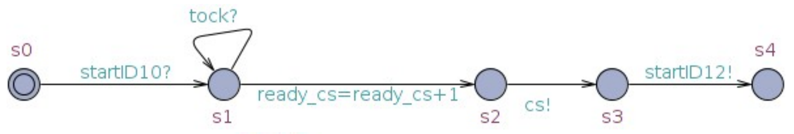
tock?
s0
startID9?
ready_cs= ready_cs+1
cs?
startID11!
s4
s1
s2
s3

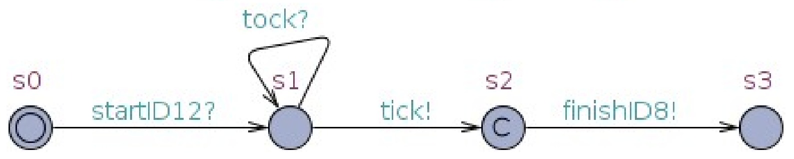


tock?
s0
startID11?
s1
tick!
s2
C
finishID7!
s3



tock?
s0
startID8?
c3!
startID10!
s3
s1
s2

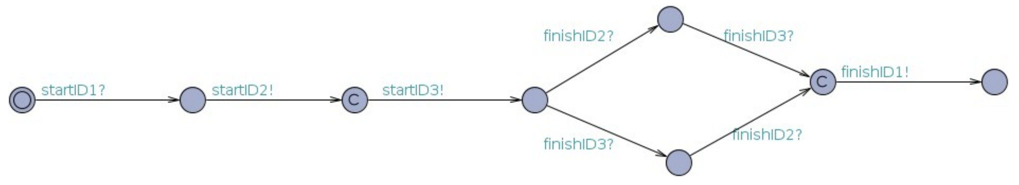




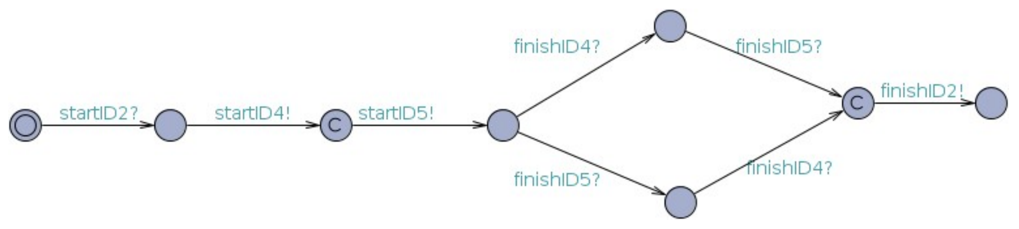
```
        ]
```

Example D.2. $(P[|\{cs\}|]Q1)[|\{\emptyset\}|]R1$

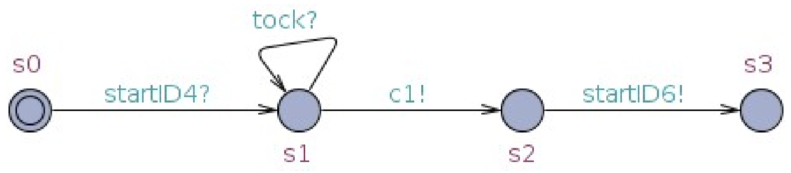
```
transform((P1[|{cs}|]Q)[|{ ∅ }|]R1))
  = transTA((P1[|{cs}|]Q)[|∅|]R1), startID1, finishID1, ∅, ∅, ∅)
  = [
```



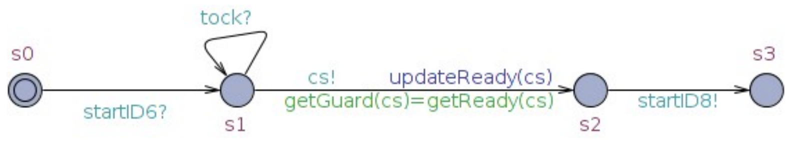
```
        ] ⌢ transTA((P1[|{cs}|]Q1), startID2, finishID2, ∅, ∅, ∅)
        ⌢ transTA(R1, startID3, finishID3, ∅, ∅, ∅)

transTA((P1[|{cs}|]Q1), startID2, finishID2, ∅, ∅, ∅)
  = [
```



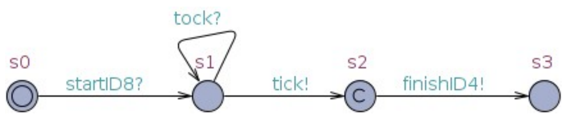
```
        ] ⌢ transTA(P1, startID4, finishID4, ∅, {cs}, ∅)
        ⌢ transTA(Q1, startID5, finishID5, {cs}, ∅, ∅)


transTA(P1, startID4, finishID4,  ∅, {cs} ∅)
  = transTA(c1->cs->SKIP, startID4, finishID4,  ∅, {cs}, ∅)
  = [
```



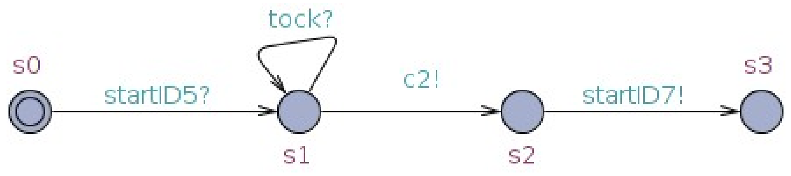
```
      ] ⌢ transTA(cs->SKIP, startID6, finishID4, ∅, {cs}, ∅)
         = [
```



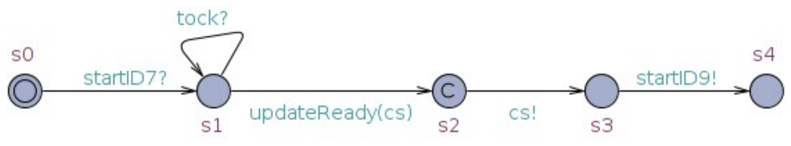
```
       ] ⌢ transTA(SKIP, startID8, finishID4, ∅, {cs}, ∅)
           = [
```



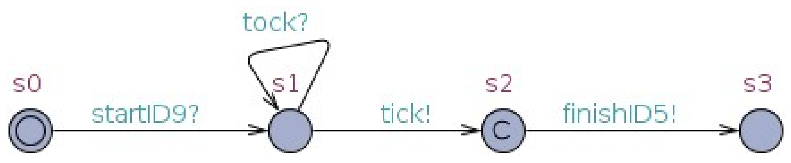
```
transTA(Q1, startID5, finishID5, {cs}, ∅, ∅)
  = transTA(c2->cs->SKIP, startID5, finishID5, {cs}, ∅, ∅)
       =[
```



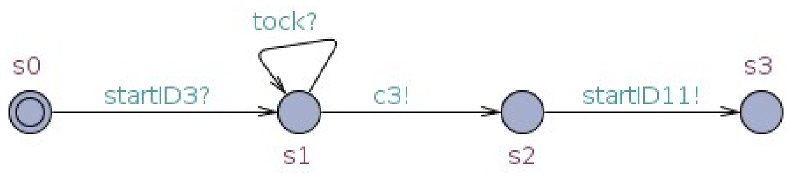
```
        ] ⌢ transTA(cs->SKIP, startID7, finishID5, {cs}, ∅, ∅)
          = [
```



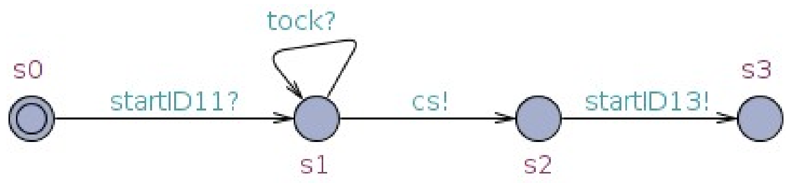
```
        ] ⌢ transTA(SKIP, startID9, finishID5, {cs}, ∅, ∅)
          = [
```



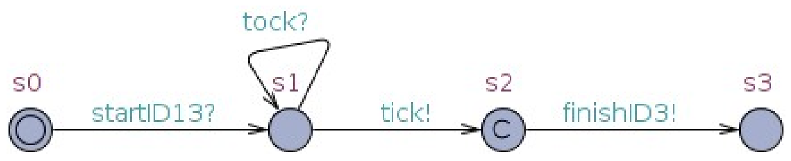
```
       ]
transTA(R1, startID3, finishID3, ∅, ∅, ∅)
 = transTA(c3->cs->SKIP, startID3, finishID3, ∅, ∅, ∅)
 = [
```



```
      ] ⌢ transTA(cs->SKIP, startID11, finishID3, ∅, ∅, ∅)
                  =[
```

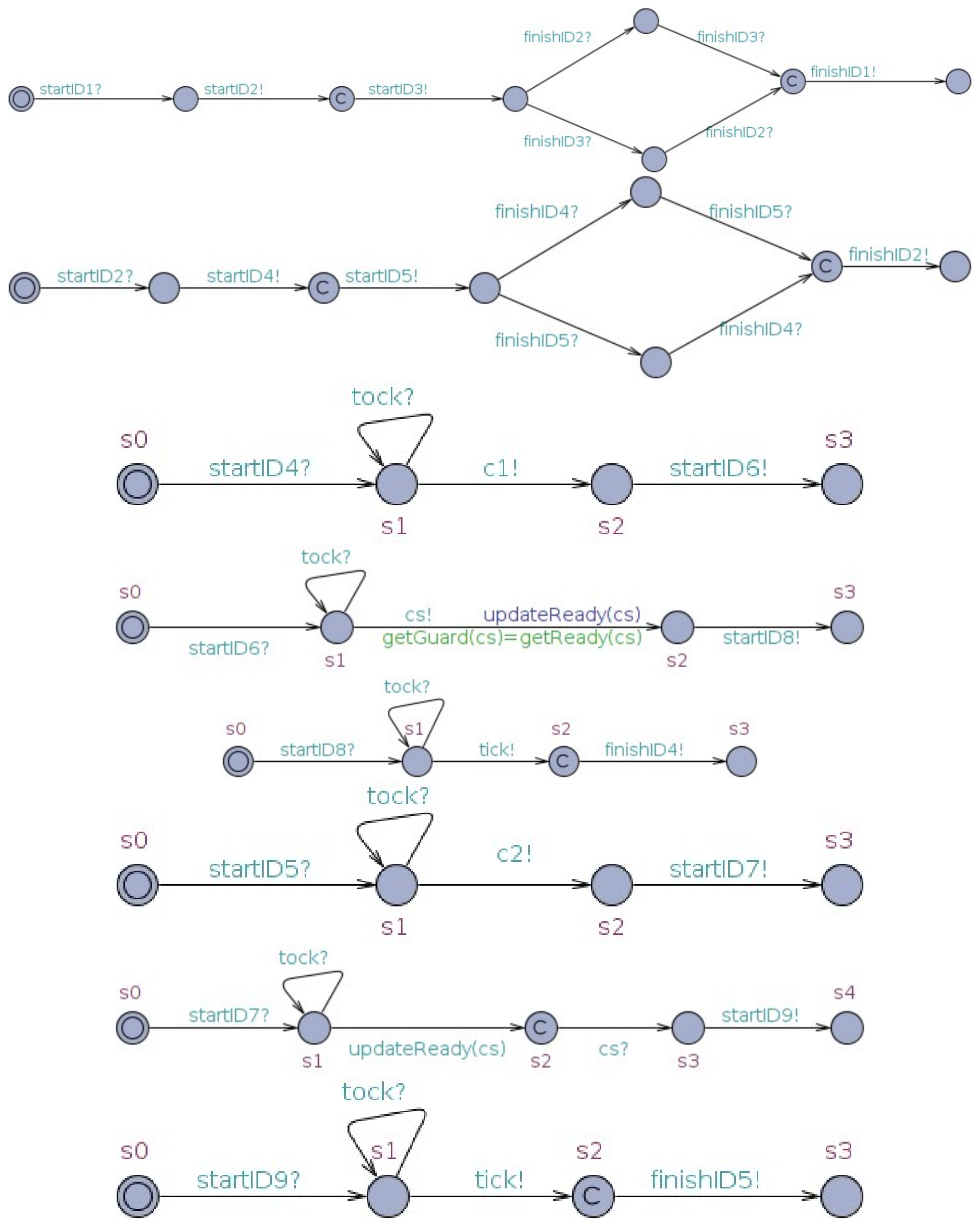


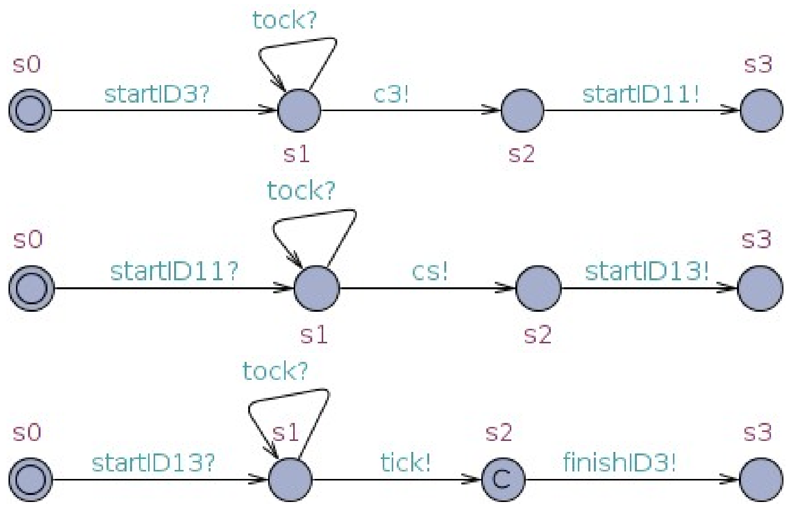
```
      ] ⌢ transTA(SKIP, startID13, finishID3, ∅, ∅, ∅)
              =[
```



```
                  ]
```

```
∴ transform((P1[|{cs}|]Q1)[|∅|]R1)
    = [
```

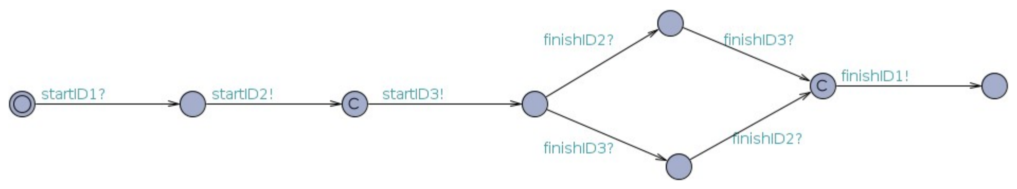




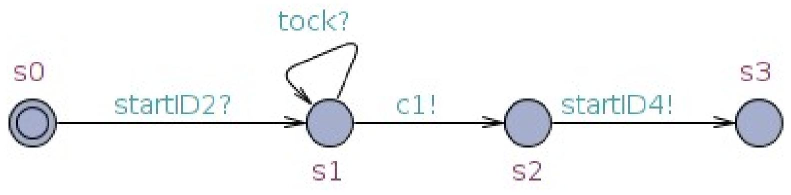
```
                ]
```

Example D.3. $P1[|\emptyset|](Q1[|\{cs\}|]R1)$

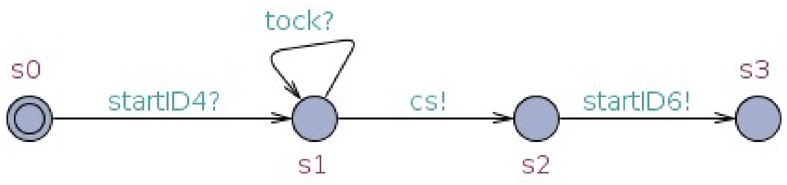
```
transform(P1[|∅|](Q1[|{cs}|]R1))
 = transTA(P1[|∅|](Q1[|{cs}|]R1), startID1, finishID1, ∅, ∅, ∅)
 = [
```



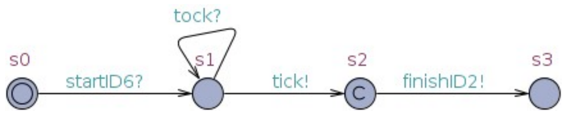
```
          ] ⌢ transTA(P1, startID2, finishID2, ∅, ∅, ∅)
          ⌢ transTA((Q1[|{cs}|]R1), startID3, finishID3, ∅, ∅, ∅)

transTA(P1, startID2, finishID2, ∅, ∅, ∅)
 = transTA(c1->cs->SKIP, startID2, finishID2, ∅, ∅, ∅)
 = [
```



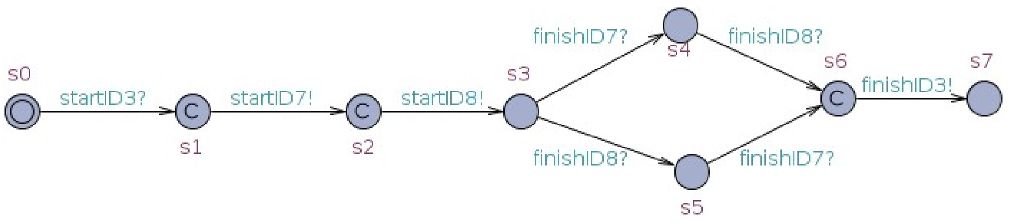
```
     ] ⌢ transTA(cs->SKIP, startID4, finishID2, ∅, ∅, ∅)
        = [
```



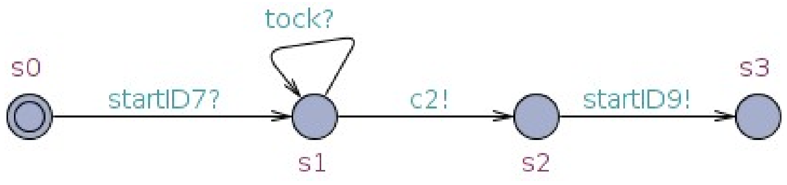
```
      ] ⌢ transTA(SKIP, startID6, finishID2, ∅, ∅, ∅)
          = [
```



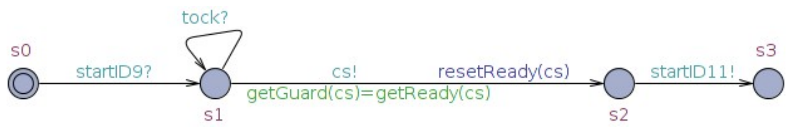
```
] 

= transTA((Q1[|{cs}|]R1), startID3, finishID3, ∅, ∅, ∅)
= [
```



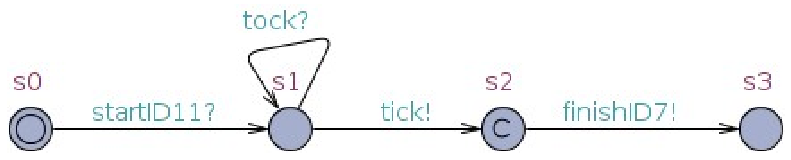
```
        ] ⌢ transTA(Q1, startID7, finishID7, ∅, {cs}, ∅)
        ⌢ transTA(R1, startID8, finishID8, {cs}, ∅, ∅)

transTA(Q1, startID7, finishID7, ∅, {cs}, ∅)
  = transTA(c2->cs->SKIP, startID7, finishID7, ∅, {cs}, ∅)
  = [
```



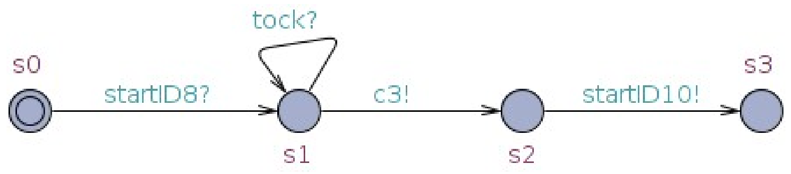
```
       ] ⌢ transTA(cs->SKIP, startID9, finishID7, ∅, {cs}, ∅)
       = [
```



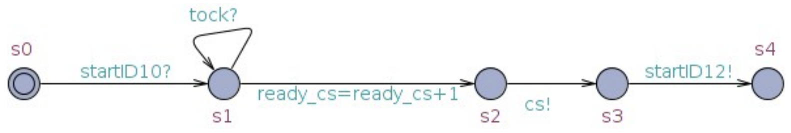
```
        ] ⌢ transTA(SKIP, startID11, finishID7, ∅, {cs}, ∅)
        = [
```



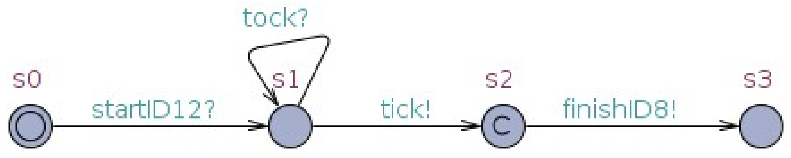
```
transTA(R1, startID8, finishID8, {cs}, ∅)
  = transTA(c3->cs->SKIP, startID8, finishID8, {cs}, ∅, ∅)
  = [
```



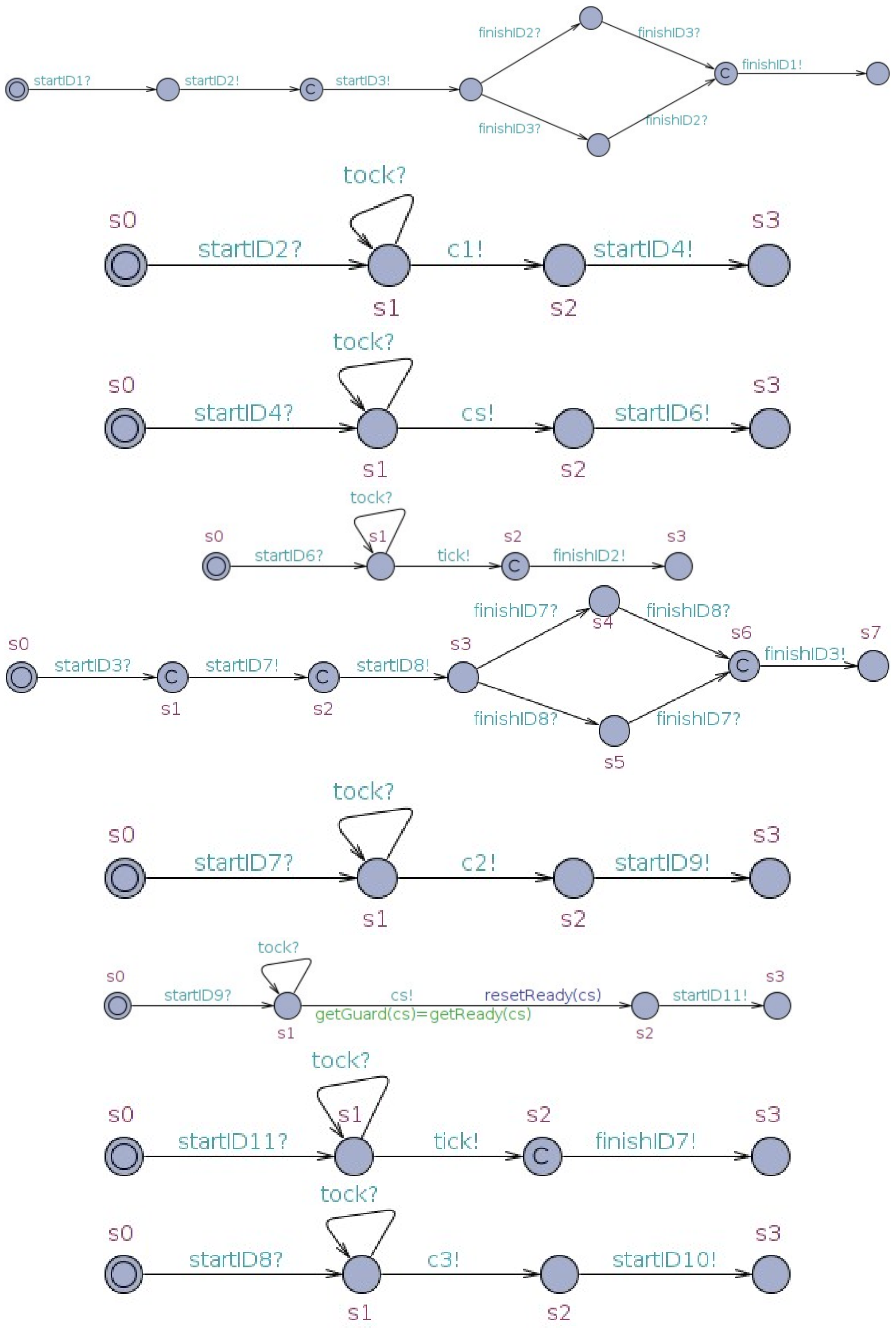
```
      ] ⌢ transTA(cs->SKIP, startID10, finishID8, {cs}, ∅, ∅)
       = [
```

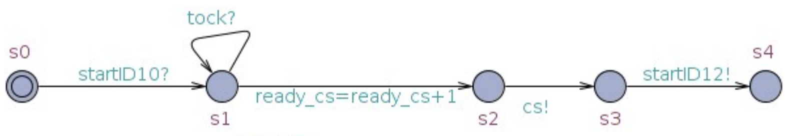


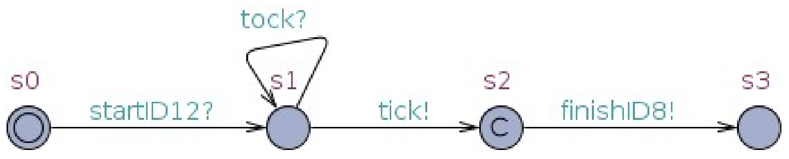
```
      ] ⌢ transTA(SKIP, startID12, finishID8, {cs}, ∅, ∅)
```



```
]

∴ transform(P1[|∅|](Q1[|{cs}|]R1))
  = [
```


startID1?
startID2!
startID3!
finishID2?
finishID3?
finishID3?
finishID2?
finishID1!
tock?
s0
startID2?
c1!
startID4!
s3
s1
s2
tock?
s0
startID4?
cs!
startID6!
s3
s1
s2
tock?
s0
startID6?
s1
tick!
s2
finishID2!
s3
s0
startID3?
s1
startID7!
s2
startID8!
s3
finishID7?
s4
finishID8?
s6
finishID3!
s7
finishID8?
s5
finishID7?
tock?
s0
startID7?
c2!
startID9!
s3
s1
s2
tock?
s0
startID9?
s1
cs!
getGuard(cs)=getReady(cs)
resetReady(cs)
s2
startID11!
s3
tock?
s0
startID11?
s1
tick!
s2
finishID7!
s3
tock?
s0
startID8?
c3!
startID10!
s3
s1
s2

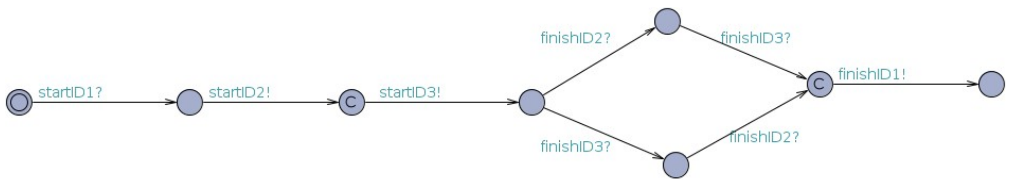




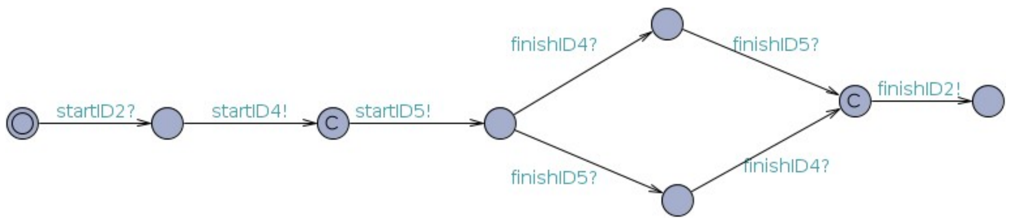
```
        ]
```

Example D.4. $(P1[|\emptyset|]Q1)[|\{cs\}|]R1$

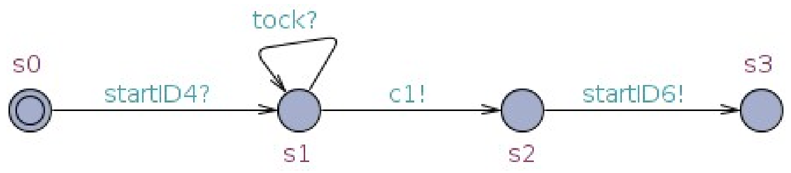
```
transform((P1[|∅|]Q1)[|{cs}|]R1)
 = transTA((P1[|∅|]Q1)[|{cs}|]R1, startID1, finishID1, ∅, ∅, ∅)
 = [
```



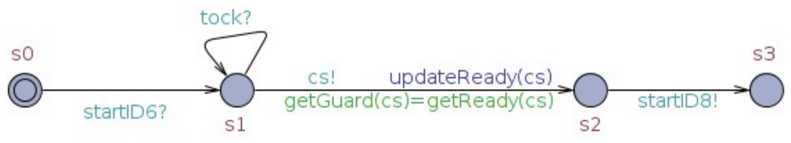
```
        ] ⌢ transTA((P1[|∅|]Q1), startID2, finishID2, {cs}, ∅, ∅)
        ⌢ transTA(R1, startID3, finishID3, {cs}, ∅, ∅)

transTA((P1[|∅|]Q1), startID2, finishID2, ∅, {cs}, ∅)
 = [
```



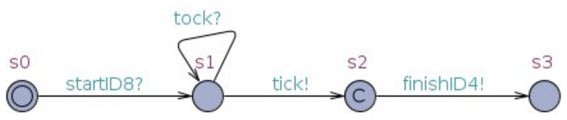
```
        ] ⌢ transTA(P1, startID4, finishID4, ∅, {cs}, ∅)
        ⌢ transTA(Q1, startID5, finishID5, ∅, {cs}, ∅)


transTA(P1, startID4, finishID4, ∅, {cs}, ∅)
 = transTA(c1->cs->SKIP, startID4, finishID4, ∅, {cs}, ∅)
 = [
```



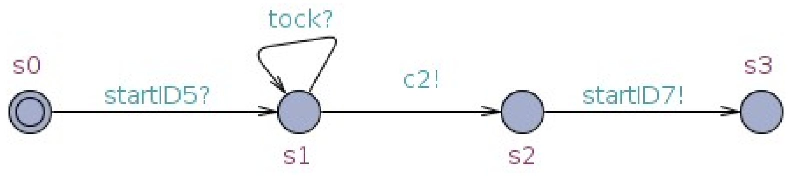
```
     ] ⌢ transTA(cs->SKIP, startID6, finishID4, ∅, {cs}, ∅)
        = [
```



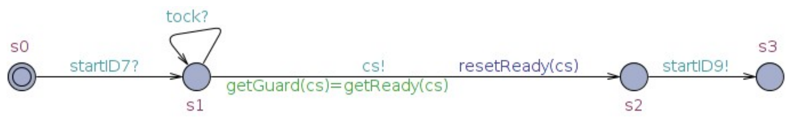
```
      ] ⌢ transTA(SKIP, startID8, finishID4, ∅, {cs}, ∅)
          = [
```



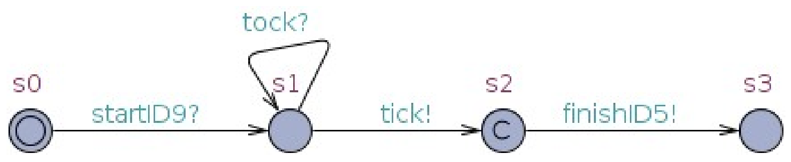
```
]

transTA(Q1, startID5, finishID5, ∅, {cs}, ∅)
 = transTA(c2->cs->SKIP, startID5, finishID5, ∅, {cs}, ∅)
      =[
```



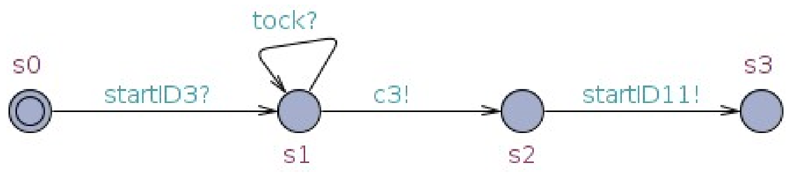
```
       ] ⌢ transTA(cs->SKIP, startID7, finishID5, ∅, {cs}, ∅)
         = [
```



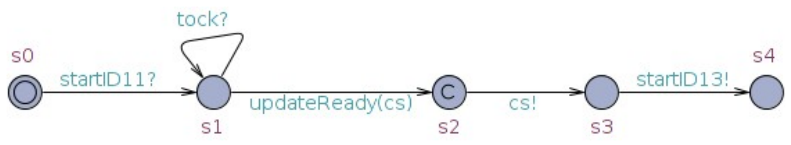
```
       ] ⌢ transTA(SKIP, startID9, finishID5, ∅, {cs}, ∅)
         = [
```



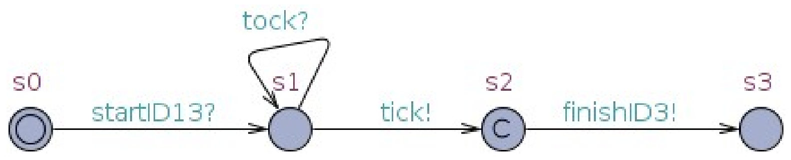
```
        ]
transTA(R1, startID3, finishID3, {cs}, ∅, ∅)
  = transTA(c3->cs->SKIP, startID3, finishID3, {cs}, ∅, ∅)
  = [
```



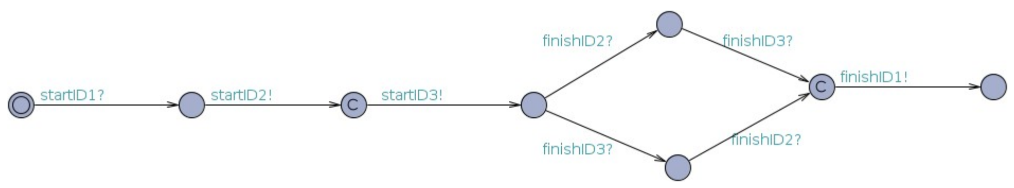
```
        ] ⌢ transTA(cs->SKIP, startID11, finishID3, {cs}, ∅, ∅)
                 =[
```

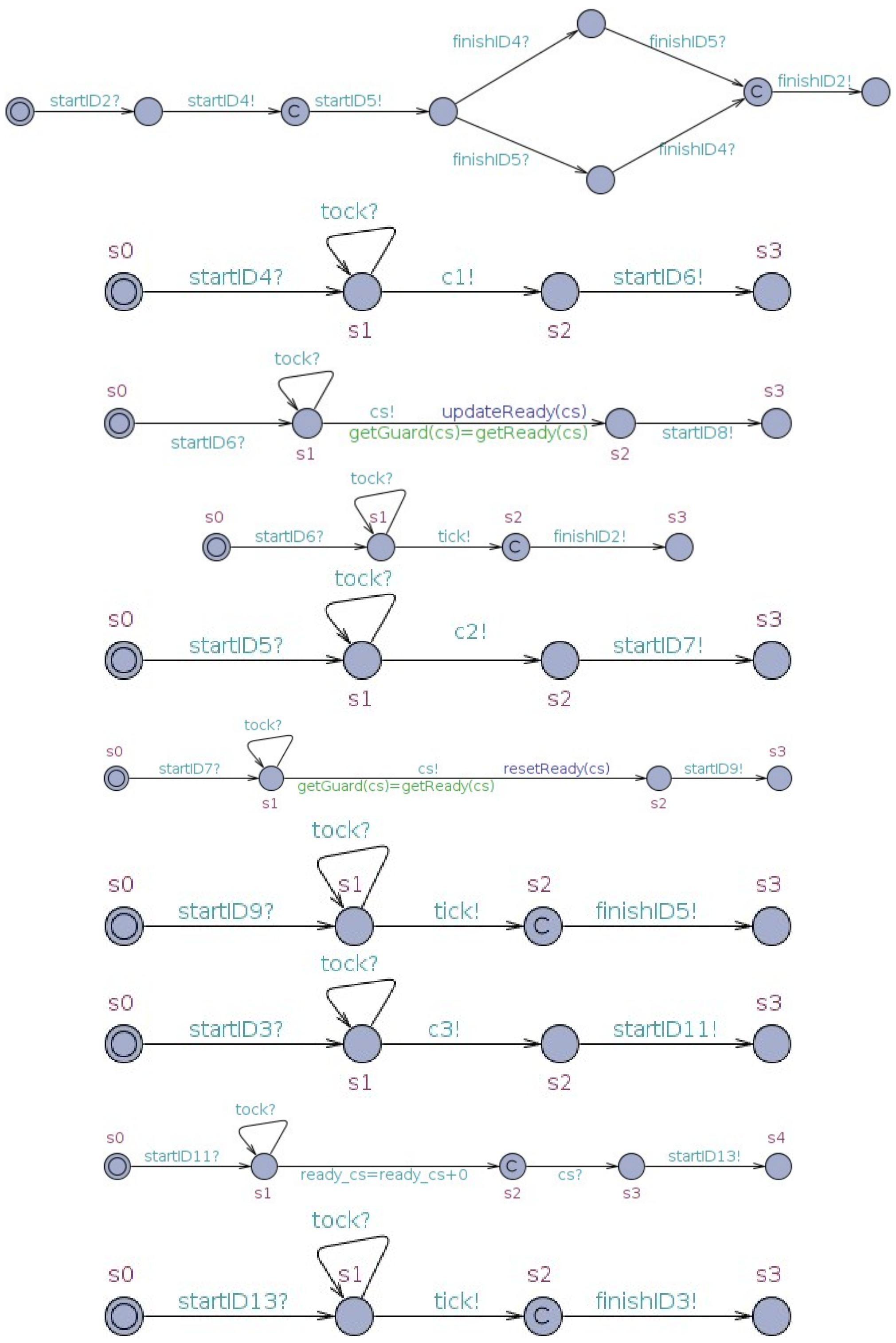


```
        ] ⌢ transTA(SKIP, startID13, finishID3, {cs}, ∅, ∅)
             =[
```



```
                 ]

∴ transform((P1[|∅|]Q1)[|{cs}|]R1)
    = [
```

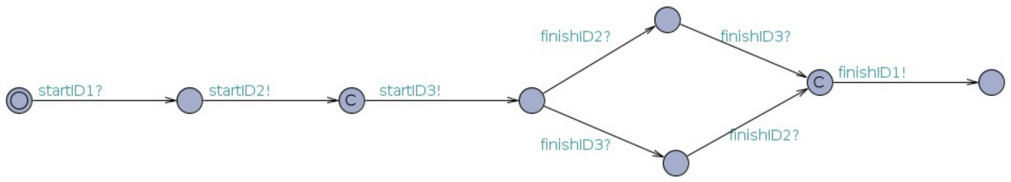



startID2?
startID4!
startID5!
finishID4?
finishID5?
finishID5?
finishID4?
finishID2!
tock?
s0
startID4?
s1
c1!
s2
startID6!
s3
tock?
s0
startID6?
s1
cs!
updateReady(cs)
getGuard(cs)=getReady(cs)
s2
startID8!
s3
tock?
s0
startID6?
s1
tick!
s2
finishID2!
s3
tock?
s0
startID5?
s1
c2!
s2
startID7!
s3
tock?
s0
startID7?
s1
cs!
resetReady(cs)
getGuard(cs)=getReady(cs)
s2
startID9!
s3
tock?
s0
startID9?
s1
tick!
s2
finishID5!
s3
tock?
s0
startID3?
s1
c3!
s2
startID11!
s3
tock?
s0
startID11?
s1
ready_cs=ready_cs+0
s2
cs?
s3
startID13!
s4
tock?
s0
startID13?
s1
tick!
s2
finishID3!
s3

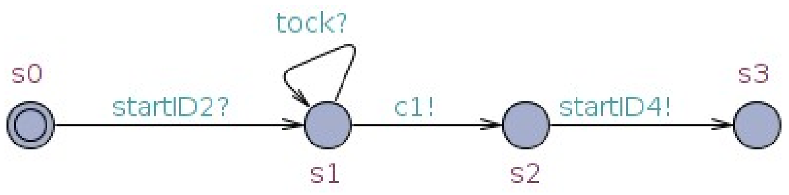
]

Example D.5. $P1[|\{cs\}|](Q1[|\{cs\}|]R1)$

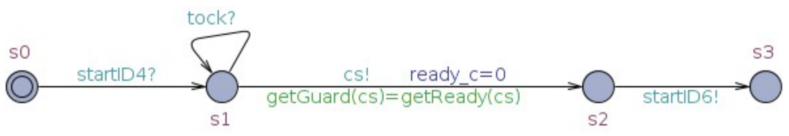
```
transform(P1[|{cs}|](Q1[|{cs}|]R1))
 = transTA(P1[|{cs}|](Q1[|{cs}|]R1), startID1, finishID1, ∅, ∅, ∅)
 = [
```



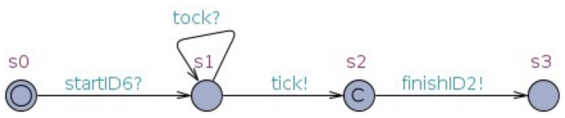
```
        ] ⌢ transTA(P1, startID2, finishID2, ∅, {cs}, ∅)
        ⌢ transTA((Q1[|{cs}|]R1), startID3, finishID3, {cs}, ∅, ∅)

transTA(P1, startID2, finishID2, ∅, {cs}, ∅)
 = transTA(c1->cs->SKIP, startID2, finishID2, ∅, {cs}, ∅)
 = [
```



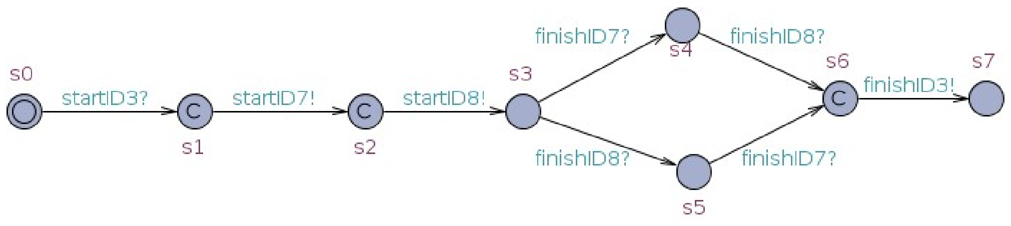
```
    ] ⌢ transTA(cs->SKIP, startID4, finishID2, ∅, {cs}, ∅)
       = [
```



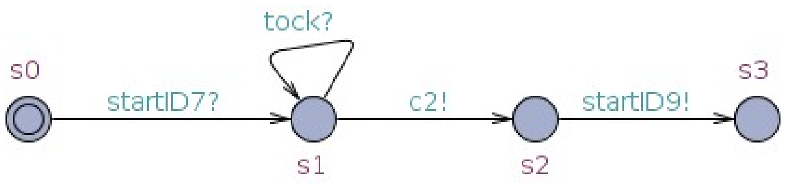
```
     ] ⌢ transTA(SKIP, startID6, finishID2, ∅, {cs}, ∅)
         = [
```



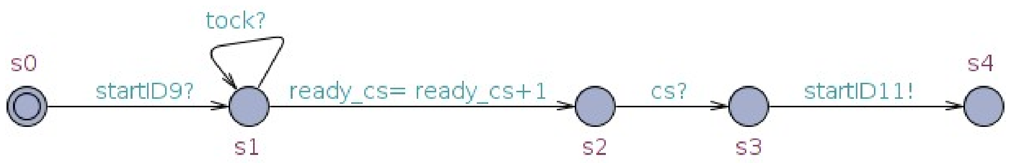
```
]
= transTA((Q1[|{cs}|]R1), startID3, finishID3, {cs}, ∅, ∅)
= [
```



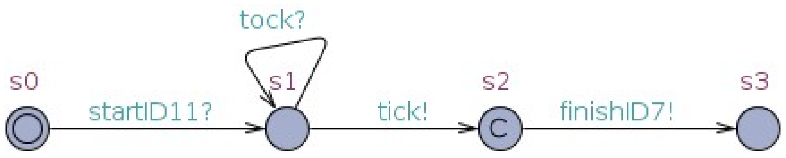
```
          ] ⌢ transTA(Q1, startID7, finishID7, {cs}, {cs}, ∅)
          ⌢ transTA(R1, startID8, finishID8, {cs}, ∅, ∅)

transTA(Q1, startID7, finishID7, {cs}, {cs}, ∅)
 = transTA(c2->cs->SKIP, startID7, finishID7, {cs}, {cs}, ∅)
 = [
```



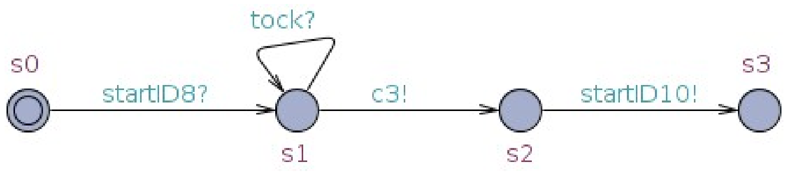
```
        ] ⌢ transTA(cs->SKIP, startID9, finishID7, {cs}, {cs}, ∅)
```



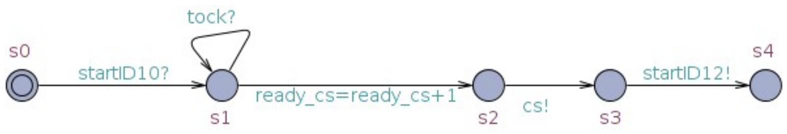
```
       ] ⌢ transTA(SKIP, startID11, finishID7, {cs}, {cs}, ∅)
```



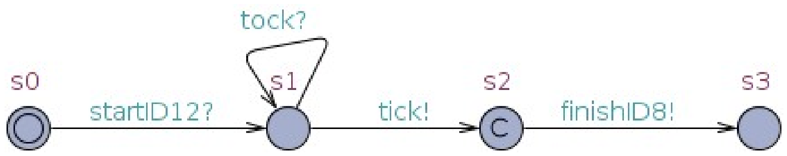
```
]
transTA(R1, startID8, finishID8, {cs}, ∅, ∅)
 = transTA(c3->cs->SKIP, startID8, finishID8, {cs}, ∅, ∅)
 = [
```



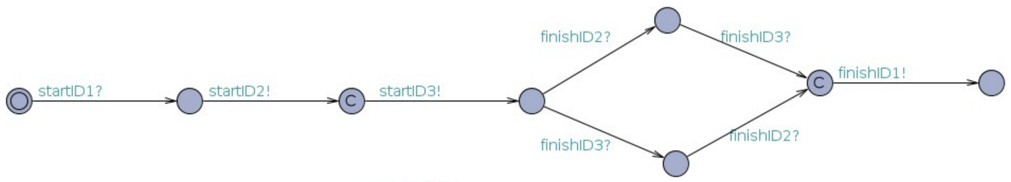
```
    ] ⌢ transTA(cs->SKIP, startID10, finishID8, {cs}, ∅, ∅)
      = [
```

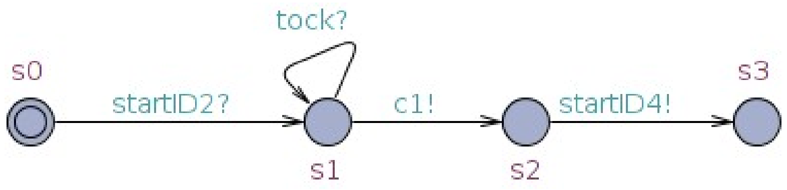


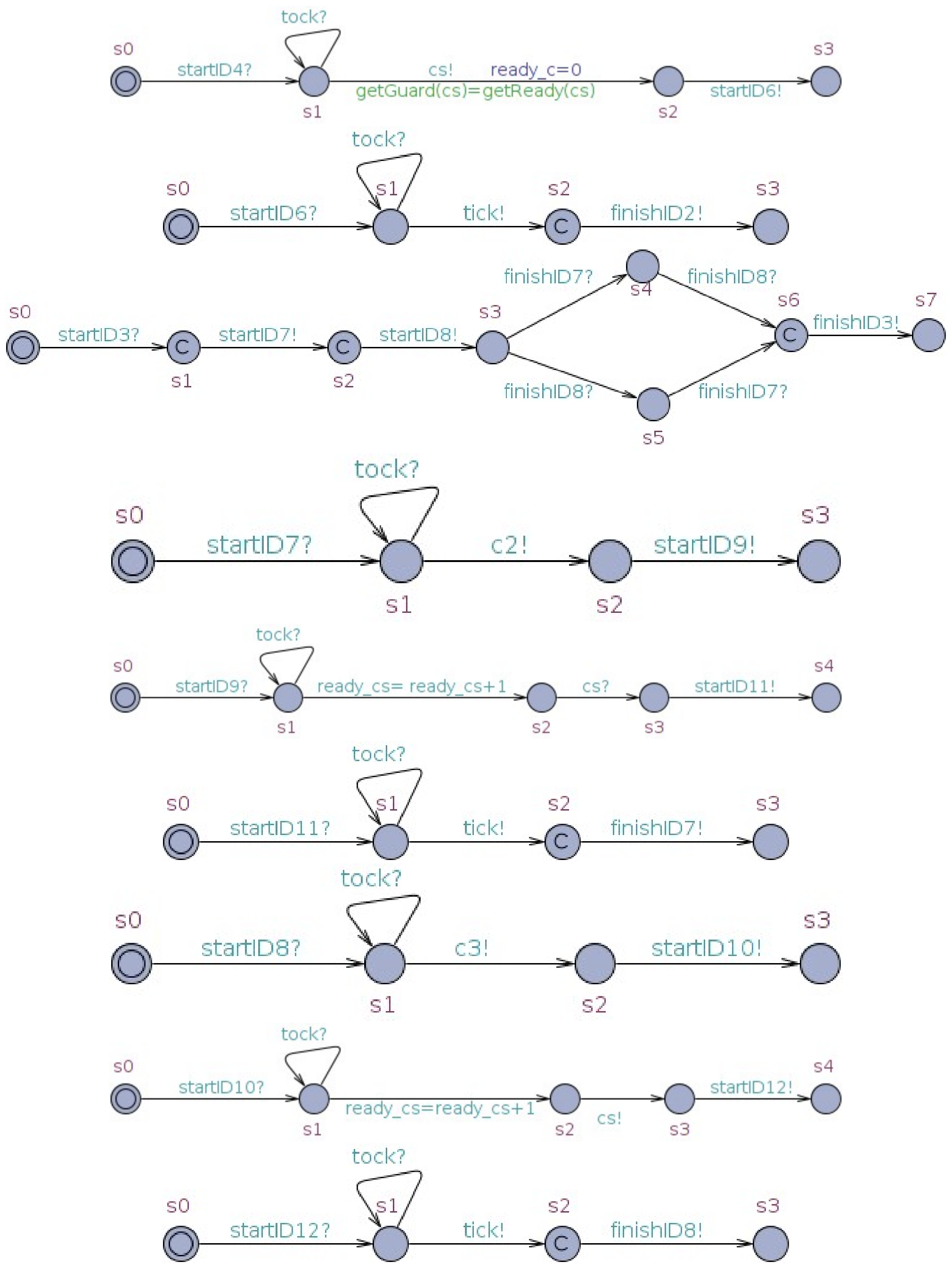
```
    ] ⌢ transTA(SKIP, startID8, finishID8, {cs}, ∅, ∅)
    = [
```



```
]

∴ transform(P1[|{cs}|](Q1[|{cs}|]R1))
  = [
```



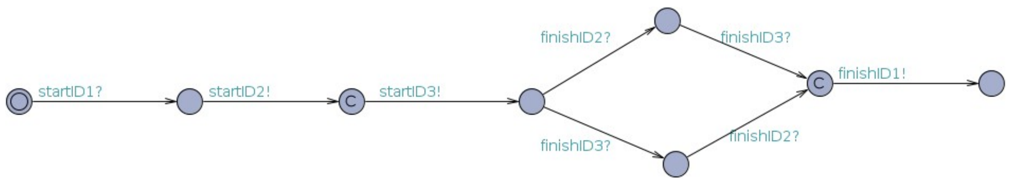



tock?
s0
startID4?
s1
cs!
ready_c=0
getGuard(cs)=getReady(cs)
s2
startID6!
s3
tock?
s0
startID6?
s1
tick!
s2
finishID2!
s3
s0
startID3?
s1
startID7!
s2
startID8!
s3
finishID7?
s4
finishID8?
s6
finishID3!
s7
finishID8?
s5
finishID7?
tock?
s0
startID7?
s1
c2!
s2
startID9!
s3
tock?
s0
startID9?
s1
ready_cs= ready_cs+1
s2
cs?
s3
startID11!
s4
tock?
s0
startID11?
s1
tick!
s2
finishID7!
s3
tock?
s0
startID8?
s1
c3!
s2
startID10!
s3
tock?
s0
startID10?
s1
ready_cs=ready_cs+1
s2
cs!
s3
startID12!
s4
tock?
s0
startID12?
s1
tick!
s2
finishID8!
s3

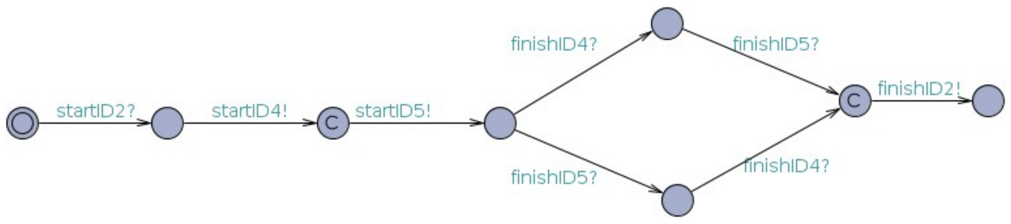
]

Example D.6. $(P1[|\{cs\}|]Q1)[|\{cs\}|]R1$

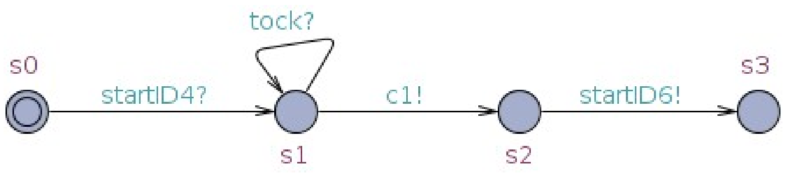
```
transform((P1[|{cs}|]Q1)[|{cs}|]R1)
 = transTA((P1[|{cs}|]Q1)[|{cs}|]R1), startID1, finishID1, ∅, ∅, ∅)
 = [
```



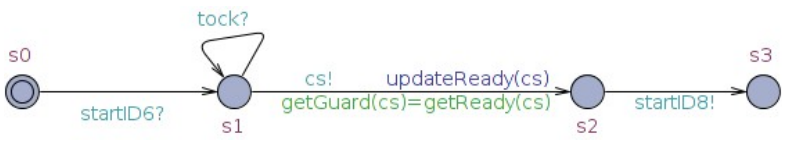
```
          ] ⌢ transTA((P1[|{cs}|]Q1), startID2, finishID2, ∅, {cs}, ∅)
          ⌢ transTA(R1, startID3, finishID3, {cs}, ∅, ∅)

transTA((P1[|{cs}|]Q1), startID2, finishID2, ∅, ∅)
 = [
```



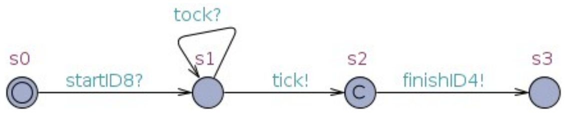
```
          ] ⌢ transTA(P1, startID4, finishID4, ∅, {cs}, ∅)
          ⌢ transTA(Q1, startID5, finishID5, {cs}, {cs}, ∅)


transTA(P1, startID4, finishID4, ∅, {cs}, ∅)
 = transTA(c1->cs->SKIP, startID4, finishID4, ∅, {cs}, ∅)
 = [
```



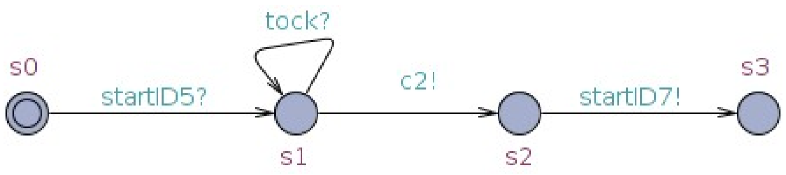
```
     ] ⌢ transTA(cs->SKIP, startID6, finishID4, ∅, {cs}, ∅)
        = [
```



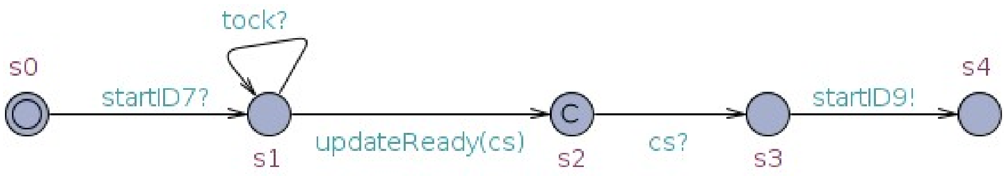
```
      ] ⌢ transTA(SKIP, startID8, finishID4, ∅, {cs}, ∅)
          = [
```



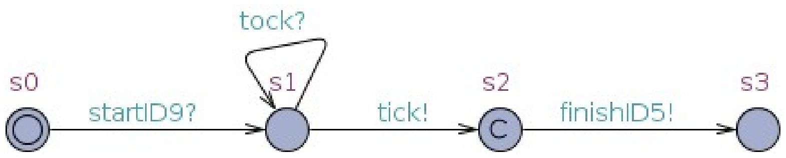
```
transTA(Q1, startID5, finishID5, {cs}, {cs}, ∅)
  = transTA(c2->cs->SKIP, startID5, finishID5, {cs}, {cs}, ∅)
       =[
```



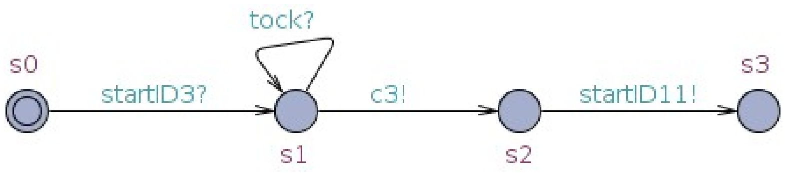
```
       ] ⌢ transTA(cs->SKIP, startID7, finishID5, {cs}, {cs}, ∅)
         = [
```



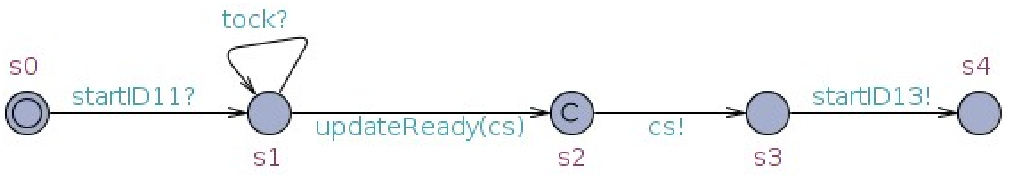
```
       ] ⌢ transTA(SKIP, startID9, finishID5, {cs}, {cs}, ∅)
         = [
```



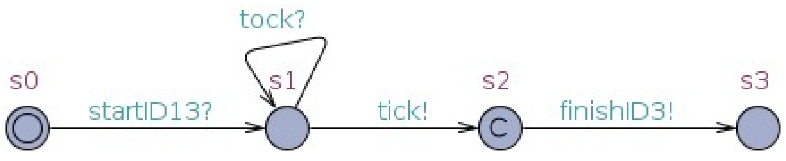
```
      ]
transTA(R1, startID3, finishID3, {cs}, ∅, ∅)
  = transTA(c3->cs->SKIP, startID3, finishID3, {cs}, ∅, ∅)
  = [
```



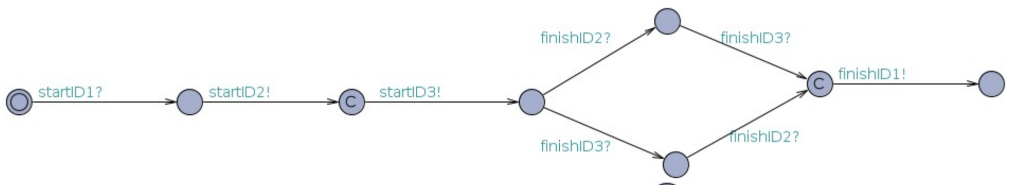
```
      ] ⌢ transTA(cs->SKIP, startID11, finishID3, {cs}, ∅, ∅)
                  =[
```

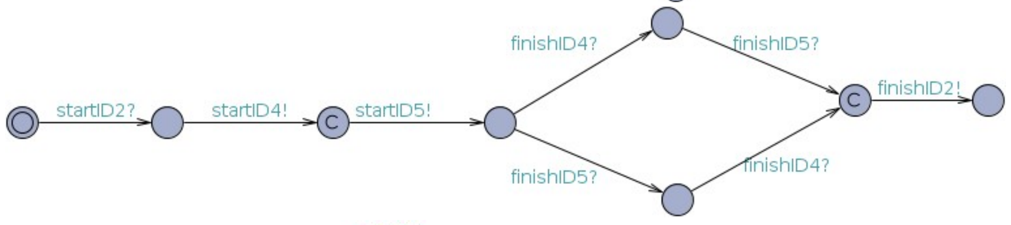


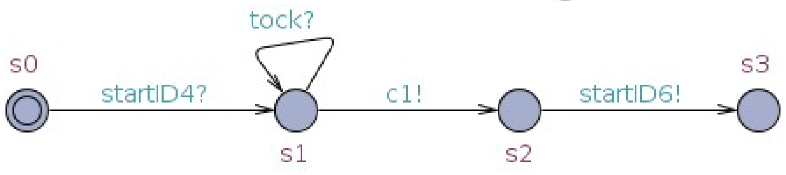
```
      ] ⌢ transTA(SKIP, startID13, finishID3, {cs}, ∅, ∅)
              =[
```

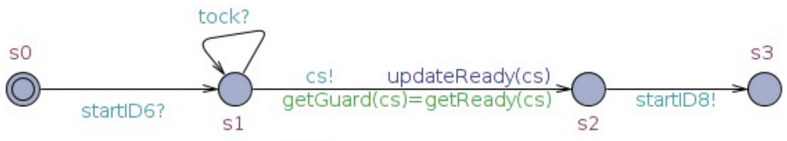


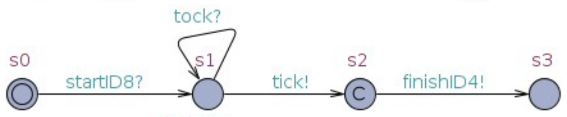
```
                  ]
 ∴ transform((P1[|{cs}|]Q1)[|{cs}|]R1)
     = [
```

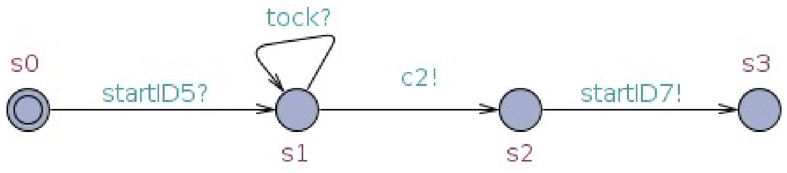


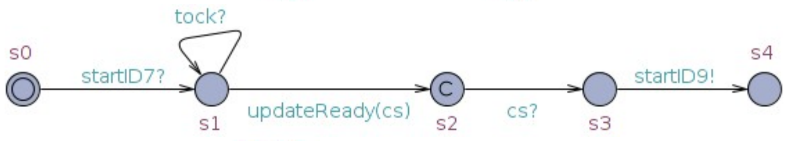


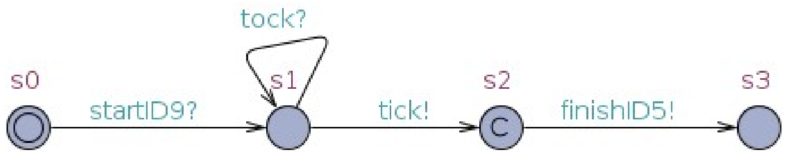


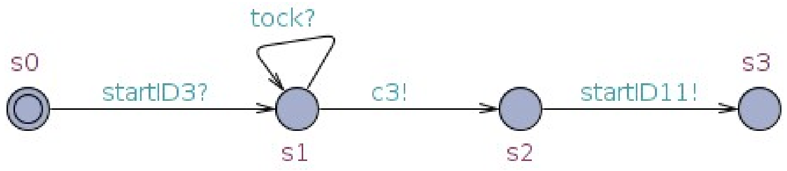

s0
startID6?
tock?
s1
cs!
updateReady(cs)
getGuard(cs)=getReady(cs)
s2
startID8!
s3

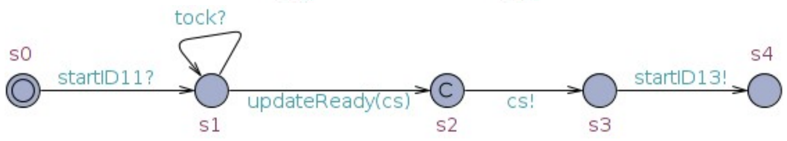

tock?
s0
startID8?
s1
tick!
s2
C
finishID4!
s3

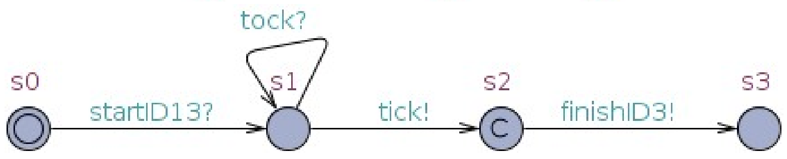

tock?
s0
startID5?
c2!
startID7!
s3
s1
s2


tock?
s0
startID7?
s1
updateReady(cs)
C
s2
cs?
s3
startID9!
s4


tock?
s0
startID9?
s1
tick!
s2
C
finishID5!
s3


tock?
s0
startID3?
c3!
startID11!
s3
s1
s2


tock?
s0
startID11?
s1
updateReady(cs)
C
s2
cs!
s3
startID13!
s4


tock?
s0
startID13?
s1
tick!
s2
C
finishID3!
s3

]

E. A Basic Tool for Checking the Steps of the Proof

This is the source code of the basic tool for automatically checking the steps of the proof, mainly formulae, syntax and terms used in the proof [16].

```haskell

{- The function pcpl is a quick-and-dirty way of revealing errors in
    a particular instance of the claim that every term in the proof is
     equal to every other term. If the claim is true, it reports "GOOD
     PROOF!", otherwise it reports that the proof is bad, and shows
    the values at each step. An incorrect step is revealed by two
    adjacent values being different. The code was developed by Jeremy
    Jacob.
-}

module ProofLayout where

import Data.List

infixr 0 :=:
data ProofLayout a = a :=: (ProofLayout a) | QED
instance Foldable ProofLayout where
  foldr f z = ffz
    where ffz QED = z
          ffz (l :=: pl) = f l (ffz pl)

pcpl :: (Eq a, Show a) => ProofLayout a -> IO ()
pcpl = putStr . report . allOK . foldr (:) []
  where
    allOK pl = (and [ l == m | (l, m) <- zip pl (tail pl)], pl)
    report (True, _ ) = "GOOD␣PROOF!\n"
    report (_   , pl) = "BAD␣PROOF!\n" ++ foldr ((++) . ("\n␣␣"++) .
        (++"\n:=:") . show) "QED\n" pl
```

[16]Thanks to Jeremy Jacob for providing this code.

F. Details of the Proof

```haskell
-- Contents
-- 1. Proof
-- 2. Definitions


module Proof_function where

import Data.List
import ProofLayout


{---------- Begining of the proof --------------------------
Proof by induction over the Natural numbers.
forall n:Nat, forall P::Proc, traces_tockCSP n P = traces_TA n .
    transTA P

BASE CASE for n=0 ------------------------------------------------
 for n=0,
    forall P::Proc, traces_tockCSP 0 P = traces_TA 0 (transTA P)
-}


proofTrans :: CSPproc -> Int -> ProofLayout [[Event]]
proofTrans    p          0   =
    traces_tockCSP 0 p
    :=: --{traces_tockCSP.0, l~>r, _0<~p}
    [[]]
    :=: --{traces_TA.0, r~>l, _0<~transTA p}
    traces_TA 0 (transTA p  ""  0  0  0 [])
    :=: QED
{- Because we have used no properties of P, other than P::CSPproc, we
     may
   conclude, by the rule of generalisation:
    forall P::Proc, traces_tockCSP 0 P = traces_TA 0 (transTA P)
-}


{- INDUCTION STEP for n>0
    ---------------------------------------------
 for n>0: forall P::Proc,
   (traces_TA n (transTA P) = traces_tockCSP n P)
   => (traces_TA (n+1) (transTA P) = traces_tockCSP (n+1) P)
-}




```

```
-- Base case: STOP
-- (traces_tockCSP n STOP = traces_TA n (transTA STOP))
--    => (traces_tockCSP (n+1) STOP = traces_TA (n+1) (transTA STOP))
proofTrans    STOP        n   =
    traces_tockCSP (n+1) STOP
    :=: --{traces_tockCSP.stop, n<~n+1}
    [(replicate l "tock") | l <- [0..n+1]]
    :=: --{List comprehensions }
    [(replicate l "tock") | l <- [0..n]] ++ [replicate (n+1) "tock"]
    :=: --{traces_tockCSP.stop}
    (traces_tockCSP n STOP) ++ [replicate (n+1) "tock"]
    :=: --{Induction hypothesis}
    nub ((traces_TA n (transTA STOP "ta" 0 0 0 [])) ++ [replicate (n
       +1) "tock"])
    :=: --{transTA.stop, l~>r}
    nub (traces_TA n taSTOP
         ++ [replicate (n+1) "tock"])
    :=: --{tracesTA.n, l~>r, tas<~STOP}
    nub ([t \\ ["startID0_0"] |
           t <- (traces'TA n taSTOP ) ]
         ++ [replicate (n+1) "tock"] )
    :=: --{traces'TA.stop, l~>r}
    [t \\ flowActions taSTOP |t <- ([("startID0_0"):s
                              |s <- [(replicate l "tock")|l <- [0..n
                                 ] ] ] ) ]
    ++ [replicate (n+1) "tock"]
    :=: --{Introducing the connection action}
    [t \\ flowActions taSTOP | t <- ([("startID0_0"):s
                               |s <- [(replicate l "tock")|l <- [0..
                                  n] ] ])  ]
    ++ [ t \\ ["startID0_0"] | t <- [["startID0_0"] ++ (replicate (n
       +1) "tock")] ]
    :=: --{List comprehensions}
    [t \\ flowActions taSTOP | t <- ([("startID0_0"):s
                                 |s <- [(replicate l "tock")|l <-
                                    [0..n+1] ] ] ) ]
    :=: --{traces'TA.stop, r~>l}
    nub ([t \\ flowActions taSTOP
          | t <- (traces'TA (n+1) taSTOP ) ] )
    :=: --{tracesTA.n, r~>l, n<~n+1, tas<~taSTOP}
    nub (traces_TA  (n+1)  taSTOP )
    :=: --{trans.stop, r->l}
    nub (traces_TA   (n+1)  (transTA STOP "ta" 0 0 0 []))
    :=: QED
        where
           taSTOP = [([ "s0", "s1"], "s0",  ["ck"], ["startID0_0"], [
               "tock"],
                        [("s0", "startID0_0", [], [], "s1"), ("s1", "
                           tock", "ck<=1", "ck", "s1")], [] )]
```

```
-- Therefore, traces_tockCSP (n+1) STOP = traces_TA  (n+1)  (transTA
    STOP)




--  Base case: SKIP
--  (traces_tockCSP n SKIP = traces_TA n (transTA SKIP))
--   => (traces_tockCSP (n+1) SKIP = traces_TA (n+1) (transTA SKIP))
proofTrans    SKIP        n   =
     sort (traces_tockCSP (n+1) SKIP)
     :=: --{traces_tockCSP.skip, n<~n+1}
     sort ([(replicate l "tock") | l <- [0..n+1] ] ++
     [(replicate l "tock")++["tick"] | l <- [0..n] ] )
     :=: --{List comprehensions }
     sort ([(replicate l "tock") | l <- [0..n] ]
              ++ [(replicate l "tock")++["tick"] | l <- [0..(n-1)]]
              ++ [replicate (n+1) "tock"] ++ [(replicate n "tock")++["
                  tick"]]  )
     :=: --{traces_tockCSP.skip}
     sort (traces_tockCSP n SKIP
             ++ [ replicate (n+1) "tock"]
             ++ [(replicate  n    "tock")++["tick"]]  )
     :=: --{Induction hypothesis}
     sort (nub (traces_TA n (transTA SKIP  "ta" 0 0 0 [])
                   ++ [ replicate (n+1) "tock"]
                   ++ [(replicate  n    "tock")++["tick"]] ) )
     :=: --{transTA.skip, l~>r}
     sort (nub (traces_TA n taSKIP
                   ++ [ replicate (n+1) "tock"]
                   ++ [(replicate  n    "tock")++["tick"]] ) )
     :=: --{tracesTA.n, l~>r, tas<~SKIP}
     sort (nub ([t \\ ["startID0_0"] | t <- (traces'TA n taSKIP ) ]
                    ++ [ replicate (n+1) "tock"]
                    ++ [(replicate  n    "tock")++["tick"]] ) )
     :=: --{traces'TA.skip, l~>r}
     sort ([t \\ flowActions taSKIP |t <- ([("startID0_0"):s | s <-
         ([(replicate l "tock")|l <- [0..n] ]
                     ++ [(replicate l "tock") ++ ["tick"]|l <- [0..(n-1)
                         ] ] ) ] ) ]
          ++ [replicate (n+1) "tock"] ++ [(replicate n "tock") ++ ["
              tick"]]   )
     :=: --{Introducing the connection action }
     sort ([t \\ flowActions taSKIP
                   |t <- ([("startID0_0"):s
                        |s <- ([(replicate l "tock")|l <- [0..n] ]
                   ++ [(replicate l "tock") ++ ["tick"]|l <- [0..(n-1)]
                       ] ) ] ) ]
          ++ [ t \\ ["startID0_0"]
                   | t <- ([["startID0_0"] ++ (replicate (n+1) "tock
                         ")] ++
                              [["startID0_0"] ++ (replicate  n      "
                                 tock") ++ ["tick"]]) ] )
    :=: --{List comprehensions}
    sort [t \\ flowActions taSKIP
              |t <- ([("startID0_0"):s
                              |s <- ([(replicate l "tock")|l <- [0..n
                                 +1] ]  ++
                                     [(replicate l "tock") ++ ["tick"]|
                                        l <- [0..n] ] ) ] ) ]
    :=: --{traces'TA.skip, r~>l}
    sort (nub ([t \\ flowActions taSKIP
           | t <- (traces'TA (n+1) taSKIP )  ]))
    :=: --{tracesTA.n, r~>l, n<~n+1, tas<~taSKIP}
    sort (nub (traces_TA  (n+1)  taSKIP   ))
    :=: --{trans.skip, r->l}
    sort (nub (traces_TA  (n+1)  (transTA SKIP "ta" 0 0 0 [])))
    :=: QED
        where taSKIP = [(["s0", "s1", "s2"], "s0",  ["ck"], ["
            startID0_0"], ["tock", "tick"],
                            [("s0", "startID0_0", [], [], "s1"),
                            ("s1", "tock", "ck<=1", "ck", "s1"),
                            ("s1", "tick", [], [], "s2")], [] ) ]
-- Therefore, traces_tockCSP (n+1) SKIP = traces_TA  (n+1)  (transTA
    SKIP)




{-- Internal choice ---------------------------------------------
Induction steps, proof for internal choice is as follows:
    traces_tockCSP (n+1) (P |~| Q) == traces_TA (n+1) (trans (P |~|Q)
        )
assuming
    traces_tockCSP n (P |~| Q) == traces_TA n (trans (P |~|Q))
    traces_tockCSP (n+1) P == traces_TA (n+1) (trans P)
    and traces_tockCSP (n+1) Q == traces_TA (n+1) (trans Q)
-}

proofTrans (IntChoice p1 p2) n =
    nub (traces_tockCSP (n+1) (IntChoice p1 p2) )
    :=: --{traces_tockCSP.internalChoice, n<~n+1}
    nub ((traces_tockCSP (n+1) p1) ++ (traces_tockCSP (n+1) p2))
    :=: --{Induction hypothesis }
    nub ((traces_TA  (n+1)  (transTA p1 "ta1" 0 0 0 []))   ++
        (traces_TA (n+1) (transTA p2 "ta2" 0 0 0 [])))
    :=: -- {tracesTA.n, l~>r}
    nub ([t \\ (flowActions (transTA p1 "ta1" 0 0 0 []))
        |t <- (traces'TA (n+1) (transTA p1 "ta1" 0 0 0 []))] ++
         [t \\ (flowActions (transTA p2 "ta2" 0 0 0 []))
        |t <- (traces'TA (n+1) (transTA p2 "ta2" 0 0 0 []))] )
    :=: -- {lemma1, l~>r}
    nub ([t \\ (flowActions (transTA p1 "ta1" 0 0 0 []))
         |t <- (traces'TA (n+1) (transTA p1 "ta1" 0 0 0 []))] ++
         [t \\ (flowActions (transTA p2 "ta2" 0 0 0 []))
         |t <- (traces'TA (n+1) (transTA p2 "ta2" 0 0 0 []))] ++
         [t \\ (flowActions taIntChoice )
             |t <- (traces'TA (n+1) taIntChoice ) ])
    :=: --{List comprehension}
    nub [t \\ (flowActions ((transTA p1 "ta1" 0 0 0 []) ++
                             (transTA p2 "ta2" 0 0 0 []) ++
                              taIntChoice ) )
                |t <- (traces'TA (n+1)
                            ((transTA p1 "ta1" 0 0 0 []) ++
                            ( transTA p2 "ta2" 0 0 0 []) ++
                              taIntChoice  ) )
        ]
    :=: --{tracesTA.n, r~>l}
    nub (traces_TA (n+1)
             ((transTA p1 "ta1" 0 0 0 []) ++
              (transTA p2 "ta2" 0 0 0 []) ++
              taIntChoice  ))
    :=: --{transTA.internalChoice, r~>l}
    nub ( traces_TA (n+1) (transTA  (IntChoice p1 p2) "ta" 0 0 0 [] )
        )
    :=: QED
        where
            taIntChoice = [(["s0", "s1", "s2", "s3"], "s0",  [],
                    ["startID0_0", "startID0_1", "startID1_1"], [],
                    [("s0", "startID0_0", [], [], "s1"),
                     ("s1", "startID0_1", [], [], "s2"),
                     ("s1", "startID1_1", [], [], "s3"),
                     ("s2", [], [], [], "s0"),
                     ("s3", [], [], [], "s0") ], [] ) ]
-- Therefore, traces_tockCSP n+1 (IntChoice p1 p2) = traces_TA n+1 (
    IntChoice p1 p2)
```

```
{-- External choice ------------------------------------------------
Induction steps, proof for internal choice is as follows:
    traces_tockCSP (n+1) (P [] Q) == traces_TA (n+1) (trans (P [] Q))
assuming
    traces_tockCSP n (P [] Q) == traces_TA n (trans (P [] Q))
    traces_tockCSP (n+1) P == traces_TA (n+1) (trans P)
    and traces_tockCSP (n+1) Q == traces_TA (n+1) (trans Q)
-}


proofTrans (ExtChoice p1 p2) n =
    nub (traces_tockCSP (n+1) (ExtChoice p1 p2) )
    :=: --{traces_tockCSP.externalChoice, n<~n+1}
    nub ((traces_tockCSP (n+1) p1) ++ (traces_tockCSP (n+1) p2))
    :=: --{Induction hypothesis }
    nub ((traces_TA  (n+1)  (transTA p1 "ta1" 0 0 0 []))   ++   (
       traces_TA (n+1) (transTA p2 "ta2" 0 0 0 [])))
    :=: --{tracesTA.n, l~>r}
    nub ([t \\ (flowActions (transTA p1 "ta1" 0 0 0 []))
            | t <- (traces'TA (n+1) (transTA p1 "ta1" 0 0 0 []))] ++
         [t \\ (flowActions (transTA p2 "ta2" 0 0 0 []))
            | t <- (traces'TA (n+1) (transTA p2 "ta2" 0 0 0 []))] )
    :=: --{Adding empty list}
    nub ([t \\ (flowActions (transTA p1 "ta1" 0 0 0 []))
        | t <- (traces'TA (n+1) (transTA p1 "ta1" 0 0 0 []))] ++
        [t \\ (flowActions (transTA p2 "ta2" 0 0 0 []))
        | t <- (traces'TA (n+1) (transTA p2 "ta2" 0 0 0 []))] ++ [] )
    :=: --{lemma2, l~>r}
    nub ([t \\ (flowActions (transTA p1 "ta1" 0 0 0 []))
        | t <- (traces'TA (n+1) (transTA p1 "ta1" 0 0 0 []))] ++
         [t \\ (flowActions (transTA p2 "ta2" 0 0 0 []))
        | t <- (traces'TA (n+1) (transTA p2 "ta2" 0 0 0 []))] ++
         [t \\ (flowActions taExtChoice )
             |t <- (traces'TA (n+1) taExtChoice ) ])
    :=: --{List comprehension}
    nub [t \\ (flowActions ((transTA p1 "ta1" 0 0 0 []) ++
                             (transTA p2 "ta2" 0 0 0 []) ++
                              taExtChoice ) )
                |t <- (traces'TA (n+1)
                            ((transTA p1 "ta1" 0 0 0 []) ++
                             (transTA p2 "ta2" 0 0 0 []) ++
                              taExtChoice ) ) ]
    :=: --{tracesTA.n, r~>l}
    nub (traces_TA (n+1)
             (taExtChoice ++ (transTA p1 "ta1" 0 0 0 []) ++
             (transTA p2 "ta2" 0 0 0 [])  ))
    :=: --{transTA.externalChoice, r~>l}
    nub ( traces_TA (n+1)
                    (transTA  (ExtChoice p1 p2) "ta3" 0 0 0 [] ) )
    :=: QED
        where
          taExtChoice =
                  [([ "s0", "s1", "s2", "s3"],    "s0",  [],
                    ["startID0_0", "startID0_1", "startID1_1"], [],
                    [("s0", "startID0_0", [], [], "s1"),
                     ("s1", "startID0_1", [], [], "s2"),
                     ("s2", "startID1_1", [], [], "s3"),
                     ("s3",  [],          [], [], "s0") ], [] ) ]
-- Thus, traces_tockCSP n+1 (ExtChoice p1 p2) = traces_TA n+1 (
    ExtChoice p1 p2)




{-- Sequential composition  -----------------------------------
Induction steps, proof for sequential composition is as follows:
    traces_tockCSP (n+1) (P ; Q) == traces_TA (n+1) (trans (P ; Q))
assuming
    traces_tockCSP n (P ; Q) == traces_TA n (trans (P ; Q))
    traces_tockCSP (n+1) P == traces_TA (n+1) (trans P)
    and traces_tockCSP (n+1) Q == traces_TA (n+1) (trans Q)
-}

proofTrans (Seq p1 p2) n =
    traces_tockCSP (n+1) (Seq p1 p2)
    :=: --{traces_tockCSP.seq, n<~n+1}
    (traces_tockCSP (n+1) p1)
    ++ [(iinit s) ++ t
        | s <- (traces_tockCSP (n+1) p1),  (ilast s) == "tick",
          t <- (traces_tockCSP (n - size_p1) p2) ]
    :=: --{Induction hypothesis }
    nub ( (traces_TA (n+1) (transTA p1 "ta1" 0 0 0 []) )
            ++ [(iinit t1) ++ t2
                | t1 <- (traces_TA
                          (n+1)
                          (transTA p1 "ta1" 0 0 0 []) ),
                  t2 <- (traces_TA
                          (n - size_p1)
                          (transTA p2 "ta2" 0 0 0 []) ) ]  )
    :=: --{tracesTA.n, l~>r}
    nub ( [t \\ (flowActions (transTA p1 "ta1" 0 0 0 []))
           | t <- (traces'TA (n+1) (transTA p1 "ta1" 0 0 0 []))]
           ++ [t \\  (flowActions ((transTA p1 "ta1" 0 0 0 [])
                   ++ (transTA p2 "ta2" 0 0 0 []) ))
           |t <-  [(iinit t1) ++ t2
                | t1 <-  traces'TA
                          (n+1)
                          (transTA p1 "ta1" 0 0 0 [] ),
                              (ilast t1) == "tick",
                   t2 <-  traces'TA
                              (n - size_p1)
                              (transTA p2 "ta2" 0 0 0 [] ) ] ]     )
    :=: --{lemma3, l~>r}
    nub ( [t \\ (flowActions ( transTA p1 "ta1" 0 0 0 []))
        | t <- (traces'TA (n+1) (transTA p1 "ta1" 0 0 0 []))]
         ++ [t \\ (flowActions ((transTA p1 "ta1" 0 0 0 [])
         ++ (transTA p2 "ta2" 0 0 0 [])) )
               |t <- [(iinit t1) ++ t2
                      | t1 <- (traces'TA (n+1)
                              (transTA p1 "ta1" 0 0 0 [] )),
                              (ilast t1) == "tick",
                        t2 <- (traces'TA (n - size_p1)
                              (transTA p2 "ta1" 0 0 0 [] ) ) ]   ]
      ++ [t \\ (flowActions taSeq ) |t <- (traces'TA (n+1) taSeq) ])
    :=: --{List comprehension}
    nub [t \\ (flowActions ((transTA p1 [] 0 0 0 []) ++
                              (transTA p2 [] 0 0 0 []) ++
                               taSeq ) )
                |t <- (traces'TA (n+1)
                             ((transTA p1 [] 0 0 0 []) ++
                              (transTA p2 [] 0 0 0 []) ++
                               taSeq
                               )
                     )
        ]
    :=: --{tracesTA.n, r~>l}
    nub (traces_TA (n+1)
              ((transTA p1 [] 0 0 0 []) ++
               (transTA p2 [] 0 0 0 []) ++
               taSeq  ))
    :=: --{transTA.seq, r~>l}
    nub ( traces_TA (n+1) (transTA  (Seq p1 p2) "ta" 0 0 0 [] ) )
    :=: QED
        where
            taSeq = [(["s0", "s1", "s2", "s3"], "s0",  [],
                        ["startID0_0", "startID0_1", "startID0_2", "
                           FinishID0", "FinishID1"], [],
                        [("s0", "startID0_0", [], [], "s1"), ("s1", "
                           startID0_1", [], [], "s2"),
                         ("s2", "FinishID1", [], [], "s3"),  ("s3", "
                            startID0_2", [], [], "s0") ], [] ) ]
            size_p1 = length ( maximum (traces_tockCSP (n+1) p1) )
{- Therefore,
   traces_tockCSP n+1 (Seq p1 p2) = traces_TA n+1 (Seq p1 p2)
-}


```

```
-- ERROR handling -------------------------------
proofTrans    _        _    =  error "to␣be␣completed..."



------ We use the following definitions in the proof -----------
--1. Definition of transTA
transTA :: CSPproc -> ProcName ->  BranchID -> StartID -> FinishID ->
     UsedNames -> [TA]   -- (node, branch, branch)
--transTA :: CSPproc -> [TA]
transTA    STOP  procName  bid  sid  fid  usedNames
           = [([ "s0", "s1"], "s0",  ["ck"], [coodAction], ["tock"],
                [("s0", coodAction, [], [], "s1"), ("s1", "tock", "ck
               <=1", "ck", "s1")], [] )]        --trans.stop
           where
            coodAction = concat ["startID", show bid, "_", show sid]

transTA  SKIP  procName  bid  sid  fid  usedNames
           = [([ "s0", "s1", "s2"], "s0",  ["ck"], [coodAction], ["
               tock", "tick"],
               [("s0", coodAction, [], [], "s1"),
                ("s1", "tock", "ck<=1", "ck", "s1"),
                ("s1", "tick", [], [], "s2")], [] ) ]   s--trans.skip
           where
            coodAction = concat ["startID", show bid, "_", show sid]

transTA    (IntChoice p1 p2)  procName  bid  sid  fid  usedNames
           =  [(["s0", "s1", "s2", "s3"], "s0",  [],
                ["startID0_0", "startID0_1", "startID1_1", "FinishID0
                   ", "FinishID1"], [],
                [("s0", "startID0_0", [], [], "s1"), ("s1", "
                   startID0_1", [], [], "s2"),
                 ("s2", "FinishID_1", [], [], "s3"), ("s3", "
                    startID1_3", [], [], "s0") ], [] ) ]
              ++ (transTA  p1  procName  bid      (sid+1)  fid
                 usedNames )
              ++ (transTA  p2  procName  (bid+1)  (sid+1)  fid
                 usedNames )
             where
               coodAction1 = concat ["startID", show bid, "_", show
                  sid]
               coodAction2 = concat ["startID", show bid, "_", show (
                  sid+1)]
               coodAction3 = concat ["startID", show (bid+1), "_",
                  show (sid+1)]                    --trans.intChoice
```

```
transTA  (ExtChoice p1 p2)  procName  bid  sid  fid  usedNames
    =  [(([ "s0", "s1", "s2", "s3"],     "s0",  [],
          [ coodAction1, coodAction2, coodAction3], [],
           [("s0", coodAction1, [], [], "s1"),
            ("s1", coodAction2, [], [], "s2"),
            ("s2", coodAction3, [], [], "s3"),
            ("s3", [],          [], [], "s0") ], [] ) ]
         ++ (transTA  p1  procName  bid      (sid+1)  fid  usedNames )
         ++ (transTA  p2  procName  (bid+1)  (sid+1)  fid  usedNames )
  where
    coodAction1 = concat ["startID", show bid,     "_", show sid]
    coodAction2 = concat ["startID", show bid,     "_", show (sid+1)]
    coodAction3 = concat ["startID", show (bid+1), "_", show (sid+1)]
                                                      --trans.extChoice

transTA  (Seq p1 p2) procName  bid  sid  fid  usedNames
    =  [(["s0", "s1", "s2", "s3"], "s0",  [],
          [coodAction0, coodAction1, coodAction2, coodAction3], [],
                [("s0", coodAction0, [], [], "s1"),
                 ("s1", coodAction1, [], [], "s2"),
                 ("s2", coodAction2, [], [], "s3"),
                 ("s3", coodAction3, [], [], "s0") ], [] ) ]
       ++ (transTA  p1  procName  bid  (sid+1)  (fid+1)  usedNames )
       ++ (transTA  p2  procName  bid  (sid+2)   fid     usedNames )
where
    coodAction0 = concat ["startID",  show bid,  "_", show  sid]
    coodAction1 = concat ["startID",  show bid,  "_", show (sid+1)]
    coodAction2 = concat ["finishID", show (fid+1)]
    coodAction3 = concat ["startID",  show bid,  "_", show (sid+2)]
                                                         --trans.seq


--2. Definition of traces_TA
traces_TA :: Int -> [TA] -> [[Event]]
traces_TA    0       _    =  [[]]            -- tracesTA.0
traces_TA    n      tas   =
                    [t \\ (flowActions tas) | t <- (traces'TA n tas)]
                                                     -- tracesTA.n


--3. Definition of traces'TA
traces'TA :: Int -> [TA]    -> [[Event]]
traces'TA    0       _       =  [[]]
traces'TA    n       []      =  [[]]
traces'TA    n    ((["s0", "s1"],  "s0",  ["ck"], [ coodAction ],
                    ["tock"], [("s0",  st, [], [], "s1"),
                    ("s1", "tock", "ck<=1", "ck", "s1")], []):tas)
            = [coodAction:s|s<-[(replicate l "tock")|l <- [0..n] ] ]
                ++ (traces'TA n tas)              -- traces'TA.stop
```

```

traces'TA    n    ((["s0", "s1", "s2"], "s0",  ["ck"], _,
                    ["tock", "tick"],
                    [("s0",  st,    [],       [],   "s1"),
                     ("s1", "tock", "ck<=1", "ck", "s1"),
                     ("s1", "tick", [],       [],   "s2")], []):tas)
            = [(replicate l "tock")|l <- [0..n] ] ++
                [(st:s ++ ["tick"])|s<-[(replicate l "tock")
                    |l <- [0..(n-1)] ] ]
                ++ (traces'TA n tas)           --traces'TA.skip

traces'TA    n    ( (["s0", "s1", "s2", "s3"], "s0",  [], _, [],
                     [("s0", coodAction1, [], [], "s1"),
                      ("s1", coodAction2, [], [], "s2"),
                      ("s1", coodAction3, [], [], "s3"),
                      ("s2", [], [], [], "s0"),
                      ("s3", [], [], [], "s0") ], [] ):tas)
            = [[coodAction1, coodAction2, coodAction3]]
              ++ (traces'TA n tas)       -- traces'TA.internalChoice

traces'TA    n    ( ([ "s0", "s1", "s2", "s3"],    "s0",  [], _, [],
                     [("s0", coodAction1, [], [], "s1"),
                      ("s1", coodAction2, [], [], "s2"),
                      ("s2", coodAction3, [], [], "s3"),
                      ("s3", [],          [], [], "s0") ], [] ):tas)
                = [[coodAction1, coodAction2, coodAction3]]
                    ++ (traces'TA n tas)
                                          --traces'TA.externalChoice

traces'TA    n    ( (["s0", "s1", "s2", "s3"], "s0",  [], _, [],
                    [("s0", coodAction0, [], [], "s1"),
                     ("s1", coodAction1, [], [], "s2"),
                     ("s2", coodAction2, [], [], "s3"),
                     ("s3", coodAction3, [], [], "s0") ], []):tas)
            = ([coodAction1, coodAction2, coodAction3])
                :(traces'TA n tas)                 -- traces'TA.seq



-- A structure for extending the proof to cover the remaining constructs
traces'TA    _   _  = [["~~Error~~"]] -- error "pending proof"

--4. Definition of traces_tockCSP
traces_tockCSP :: Int -> CSPproc -> [[Event]]
traces_tockCSP    0      _       =  [[]]        -- traces_tockCSP.0
traces_tockCSP    n      STOP
               =  [(replicate l "tock") | l <- [0..n] ]
                                              -- traces_tockCSP.stop
```

```

traces_tockCSP    n      SKIP
               =  [(replicate l "tock") | l <- [0..n] ] ++
                  [(replicate l "tock") ++ ["tick"]
                      | l <- [0..(n-1)] ]    --traces_tockCSP.skip

traces_tockCSP    n      (IntChoice p1 p2 )
              =  (traces_tockCSP n p1) ++
                 (traces_tockCSP n p2)
                              -- traces_tockCSP.internalChoice

traces_tockCSP    n      (ExtChoice p1 p2 )
              =  (traces_tockCSP n p1) ++ (traces_tockCSP n p2)
                              -- traces_tockCSP.externalChoice

traces_tockCSP    n      (Seq p1 p2 )
              =  if (n > size_p1)
                 then (traces_tockCSP n p1) ++
                    [(iinit t1) ++ t2
                        | t1 <- (traces_tockCSP n p1),
                                (ilast t1) == "tick",
                          t2 <- (traces_tockCSP (n - size_p1)  p2) ]
                 else (traces_tockCSP n p1)
        where
            size_p1 = length (maximum (traces_tockCSP (n+1) p1) )
                                             -- traces_tockCSP.seq


--5. Definition of flowActions
flowActions :: [TA]                        -> [String]
flowActions    []                          =  []
                                           -- flowActions.0
flowActions    [(_, _, _, x, _, _, _)]    =  x      -- flowActions.1
flowActions    ((_, _, _, x, _, _, _):ts)
        = concat [x, (flowActions ts)]            -- flowActions.n


{- A lemma for the binary constructors to show that the traces of the
     connecting TA are empty after removing the connections actions.
    the lemma established that if the traces ts contains a trace (like
     ts is empty in this case), so adding the same trace will not
    change the value of the traces -}

-- A lemma for the connection TA of the internal choice
lemma1 n = nub ts                                     -- lemma1
        :=:
        nub (ts ++ [t \\ (flowActions [(["s0", "s1", "s2", "s3"],
             "s0", [], ["startID0_0", "startID0_1", "startID1_1"],
                 [], [("s0", "startID0_0", [], [], "s1"),
               ("s1", "startID0_1", [], [], "s2"),
               ("s1", "startID1_1", [], [], "s3"),
               ("s2", [],           [], [], "s0"),
               ("s3", [], [], [], "s0") ], [] ) ] )
               |t <- (traces'TA n
                       [(["s0", "s1", "s2", "s3"], "s0",  [],
                       ["startID0_0", "startID0_1", "startID1_1"], [],
                       [("s0", "startID0_0", [], [], "s1"),
                        ("s1", "startID0_1", [], [], "s2"),
                        ("s1", "startID1_1", [], [], "s3"),
                        ("s2", [],           [], [], "s0"),
                        ("s3", [],           [], [], "s0") ], [] ) ]
                     ) ]
             )
         :=:
          [[]]
         :=:
          QED
             where
                 ts = [[]]



-- A lemma for the connection TA of the external choice
lemma2 n = nub ts                                        -- lemma2
     :=:
     nub (ts ++
         [t \\ (flowActions
                 [([ "s0", "s1", "s2", "s3"], "s0",  [],
                   ["startID0_0", "startID0_1", "startID1_1"], [],
                   [("s0", "startID0_0", [], [], "s1"),
                    ("s1", "startID0_1", [], [], "s2"),
                    ("s2", "startID1_1", [], [], "s3"),
                    ("s3", [], [], [], "s0") ], [] ) ] )
          |t <- (traces'TA n [([ "s0", "s1", "s2", "s3"],"s0",  [],
                    ["startID0_0", "startID0_1", "startID1_1"], [],
                    [("s0", "startID0_0", [], [], "s1"),
                     ("s1", "startID0_1", [], [], "s2"),
                     ("s2", "startID1_1", [], [], "s3"),
                     ("s3", [],           [], [], "s0") ], []) ]
                 ) ]
         )
     :=:
     [[]]
     :=:
      QED
     where
         ts = [[]]

```

```

-- A lemma for the connection TA of the sequential composition
lemma3 n = nub ts                                        -- lemma3
        :=:
        nub (ts ++
        [t \\ (flowActions
                 [(["s0", "s1", "s2", "s3"], "s0", [],
                   ["startID0_0", "startID0_1", "startID0_2", "
                      FinishID0", "FinishID1"], [],
                     [("s0", "startID0_0", [], [], "s1"),
                      ("s1", "startID0_1", [], [], "s2"),
                      ("s2", "FinishID1",  [], [], "s3"),
                      ("s3", "startID0_2", [], [], "s0") ], [] )])
                        |t <- (traces'TA n
                                [(["s0", "s1", "s2", "s3"], "s0", [],
                                ["startID0_0", "startID0_1",
                                 "startID0_2", "FinishID0",
                                 "FinishID1"], [],
                                [("s0", "startID0_0", [], [], "s1"),
                                 ("s1", "startID0_1", [], [], "s2"),
                                 ("s2", "FinishID1",  [], [], "s3"),
                                 ("s3", "startID0_2", [], [], "s0")],
                                [] ) ] ) ] )
        :=:
         [[]]
        :=:
         QED
            where
               ts = [[]]


-- ilast is similar to the function last with the capability of
    handling empty trace
ilast :: [Event] -> Event
ilast    []      =  []          -- ilast.0
ilast    xs      =  last xs     -- ilast.1


-- iinit is similar to the function init with the capability of
    handling empty trace
iinit :: [Event] -> [Event]
iinit    []      =  []          -- iinit.0
iinit    xs      =  init xs     -- iinit.1


-- list_interleave
list_interleave :: [Event] -> [Event] -> [Event]
list_interleave    []         ys      =  ys          -- interleave.0
list_interleave    (x:xs)     ys      =  (x:(list_interleave ys xs))
```

```haskell
                                                  -- interleave.1

tracesInterleave t1 t2
  = nub (concat [[x++y, (list_interleave x y)] | x <- t1, y <- t2 ] )
                                                  -- traceInterleave.1

-- Example
traceInterleave_eg
  = tracesInterleave [[], ["a"], ["a", "b"]]  [[], ["x"], ["x", "y"]]


-- Data type for the CSP process -----------------------------------
data CSPproc  =  STOP
               | SKIP
               | WAIT         Int
               | Prefix       Event     CSPproc
               | IntChoice    CSPproc   CSPproc
               | ExtChoice    CSPproc   CSPproc
               | Seq          CSPproc   CSPproc
               | Interleave   CSPproc   CSPproc
               | GenPar       CSPproc   CSPproc     [Event]
               | Interrupt    CSPproc   CSPproc
               | Exception    CSPproc   CSPproc     [Event]
               | Timeout      CSPproc   CSPproc Int
               | Hiding       CSPproc   [Event]
               | Rename       CSPproc   [(Event, Event)]
               | EDeadline    Event     Int
               | ProcID       String


type TA = ([State], State, [Clock], [Action], [Action] , [(State,
    Action, Clock, Invariant, State)], [Invariant] )

type ProcName    =  String                -- An identifier for each TA
type BranchID    =  Int                   -- An index for the braches
type StartID     =  Int                   -- An index for the start
    event
type FinishID    =  Int                   -- An index for the finish
    event
type SyncPoint   = (Event, String)        -- Assign an identifier for
    a sync point
type UsedNames   = [String]               -- List of the names used in
     developing the translation rules.
type State       =  String
type Clock       =  String
type Action      =  String
type Invariant   =  String
type Event       =  String
type NamedProc   =  String
```

References

[1] Alvaro Miyazawa, Pedro Ribeiro, Wei Li, Ana Cavalcanti, and Jon Timmis. Automatic Property Checking of Robotic Applications. *IEEE/RSJ Int. Conf. Intell. Robot. System*, 2017.

[2] Pedro Ribeiro, Alvaro Miyazawa, Wei Li, Ana Cavalcanti, Jon Timmis, and Jim Woodcock. Modelling and Verification of Timed Robotic Controllers. *Integrated Formal Methods*, 2017.

[3] Andreas Angerer, Remi Smirra, Alwin Hoffmann, Andreas Schierl, Michael Vistein, and Wolfgang Reif. A Graphical Language for Real-Time Critical Robot Commands. *Proc. Third Int. Work. Domain-Specific Lang. Model. Robot. Syst. (DSLRob 2012)*, 2012.

[4] Matt Luckcuck, Marie Farrell, Louise A Dennis, Clare Dixon, and Michael Fisher. Formal specification and verification of autonomous robotic systems: A survey. *ACM Computing Surveys (CSUR)*, 52(5):1–41, 2019.

[5] Lucas Liebenwein, Cenk Baykal, Igor Gilitschenski, Sertac Karaman, and Daniela Rus. Sampling-based approximation algorithms for reachability analysis with provable guarantees. *Proceedings of Robotics: Science and Systems 2018*, 2018.

[6] Andrew William Roscoe. *Understanding Concurrent Systems.* Springer Science & Business Media, 2010.

[7] Gerd Behrmann, Alexandre David, Kim Guldstrand Larsen, Paul Pettersson, and Wang Yi. Developing UPPAAL Over 15 Years. *Software: Practice and Experience*, 41(2):133–142, 2011.

[8] Bruno Siciliano and Oussama Khatib. *Springer handbook of robotics*. Springer, 2016.

[9] Shimon Y Nof. *Handbook of industrial robotics*. John Wiley & Sons, 1999.

[10] Robert Bogue. Strong prospects for robots in retail. *Industrial Robot: the international journal of robotics research and application*, 2019.

[11] E. Demaitre. Five robotics predictions for 2016. Robotics Business Review.” (2016)., 2016. Available at `https://tinyurl.com/y3odb9zq` (Accessed: 19th October, 2020).

[12] Selma Kchir, Saadia Dhouib, Jeremie Tatibouet, Baptiste Gradoussoff, and Max Da Silva Simoes. RobotML for industrial robots: Design and simulation of manipulation scenarios. In *IEEE Int. Conf. Emerg. Technol. Fact. Autom. ETFA*, 2016.

[13] Alvaro Miyazawa, Ana Cavalcanti, Pedro Ribeiro Wei Li, Jim Woodcock, and Jon Timmis. RoboChart Reference Manual. Technical report, University of York, 2016.

[14] Philip Koopman and Michael Wagner. Autonomous vehicle safety: An interdisciplinary challenge. *IEEE Intelligent Transportation Systems Magazine*, 9(1):90–96, 2017.

[15] S Alexandrova, Z Tatlock, and M Cakmak. RoboFlow: A flow-based visual programming language for mobile manipulation tasks. *2015 IEEE Int. Conf. Robot. Autom.*, pages 5537–5544, 2015.

[16] Hoang Pham. *Software reliability*. Springer Science & Business Media, 2000.

[17] Christian Schlegel, Andreas Steck, and Alex Lotz. Robotic software systems: From code-driven to model-driven software development. *Robotic Systems-Applications, Control and Programming*, pages 473–502, 2012.

[18] John Peterson, Paul Hudak, and Conal M. Elliott. Lambda in motion: Controlling robots with Haskell. *Lect. Notes Comput. Sci. (including Subser. Lect. Notes Artif. Intell. Lect. Notes Bioinformatics)*, 1999.

[19] Alex Brooks, Tobias Kaupp, Alexei Makarenko, Stefan Williams, and Anders Oreback. Towards component-based robotics. In *2005 IEEE/RSJ International Conference on Intelligent Robots and Systems*, pages 163–168. IEEE, 2005.

[20] Issa Nesnas, Richard Volpe, Tara Estlin, Hari Das, Richard Petras, and D Mutz. Toward developing reusable software components for robotic applications. *Proc. 2001 IEEE/RSJ Int. Conf. Intell. Robot. Syst. Expand. Soc. Role Robot. Next Millenn. (Cat. No.01CH37180)*, 2001.

[21] Saddek Bensalem, Matthieu Gallien, Félix Ingrand, Imen Kahloul, and Nguyen Thanh-Hung. Designing autonomous robots. *IEEE Robotics & Automation Magazine*, 16(1):67–77, 2009.

[22] Geoffrey Biggs and Bruce Macdonald. A Survey of Robot Programming Systems. *Proc. Australas. Conf. Robot. Autom.*, pages 1–3, 2003.

[23] Arie Van Deursen, Paul Klint, and Joost Visser. Domain-specific languages: An annotated bibliography. *ACM Sigplan Notices*, 35(6):26–36, 2000.

[24] Arne Nordmann, Nico Hochgeschwender, and Sebastian Wrede. A Survey on Domain-Specific Languages in Robotics. *J. Softw. Eng. Robot.*, 1(July):195–206, 2016.

[25] Anthony M. Sloane. *Software Abstractions: Logic, Language, and Analysis by Daniel Jackson, The MIT Press, 2006, 366pp, ISBN 978-0262101141.* 2009.

[26] Anthony Mallet and Matthieu Herrb. Recent developments of the genom robotic component generator. In *6th National Conference on Control Architectures of Robots*, pages 4–p, 2011.

[27] F Fleurey and A Solberg. A Domain Specific Modeling Language supporting Specification, Simulationand Execution of Dynamic Adaptive Systems. *Model. ACM/IEEE 12th Int. Conf. Model. Eng. Lang. Syst.*, pages 606–621, 2009.

[28] Paul Black, Mark Badger, Barbara Guttman, and Elizabeth Fong. Dramatically reducing software vulnerabilities: Report to the white house office of science and technology policy. Technical report, National Institute of Standards and Technology, 2016.

[29] John Fitzgerald, Juan Bicarregui, Peter Gorm Larsen, and Jim Woodcock. Industrial deployment of formal methods: Trends and challenges. In *Industrial Deployment of System Engineering Methods*, pages 123–143. Springer, 2013.

[30] Paul Black, Larry Feldman, Gregory Witte, et al. Dramatically reducing software vulnerabilities. Technical report, National Institute of Standards and Technology, 2017.

[31] Mohammed Foughali, Bernard Berthomieu, Silvano Dal Zilio, Félix Ingrand, and Anthony Mallet. Model checking real-time properties on the functional layer of autonomous robots. In *International Conference on Formal Engineering Methods*, pages 383–399. Springer, 2016.

[32] Edmund M Clarke and Jeannette M Wing. Formal methods: State of the art and future directions. *ACM Computing Surveys (CSUR)*, 28(4):626–643, 1996.

[33] Charles Antony Richard Hoare. Communicating sequential processes. *Communications of the ACM*, 21(8):666–677, 1978.

[34] Jim Woodcock and Jim Davies. *Using Z: Specification Refinement and Proof*. Prentice Hall International, 1996.

[35] Daniel Jackson. *Software abstractions*, volume 2. MIT press Cambridge, 2006.

[36] Rajeev Alur. *TECHNIQUES FOR AUTOMATIC VERIFICATION OF REAL-TIME SYSTEMS*. PhD thesis, 1991.

[37] Steve Schneider. Concurrent and real time systems : the CSP approach. *Worldw. Ser. Comput. Sci.*, 2010.

[38] Kun Wei, Jim Woodcock, and Ana Cavalcanti. Circus time with reactive designs. In *International Symposium on Unifying Theories of Programming*, pages 68–87. Springer, 2012.

[39] Gavin Lowe. Specification of communicating processes: temporal logic versus refusals-based refinement. *Formal Aspects of Computing*, 20(3):277–294, 2008.

[40] Edmund M. Clarke, E Allen Emerson, and A Prasad Sistla. Automatic verification of finite-state concurrent systems using temporal logic specifications. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 8(2):244–263, 1986.

[41] Alexandre Boulgakov A.W. Roscoe Thomas Gibson-Robinson, Philip Armstrong. FDR3 — A Modern Refinement Checker for CSP. In Erika Ábrahám and Klaus Havelund, editors, *Tools and Algorithms for the Construction and Analysis of Systems*, volume 8413 of *Lecture Notes in Computer Science*, pages 187–201, 2014.

[42] J.A Brzozowski and C.J.H. Seger. Advances in asynchronous circuit theory. Part I : gate and unbounded inertial delay. *Bull. Eur. Assoc. Theor. Comput. Sci. EATCS, Vol. 42(1990), No. October, p. 198-249*, 1990.

[43] Patricia Bouyer. An introduction to timed automata. 2011. A lecture note that is available at `www.lsv.fr/~bouyer/files/mpri1112.pdf` (Last accessed 21 October 2020).

[44] J Norberto Pires. *Industrial robots programming: building applications for the factories of the future*. Springer Science & Business Media, 2007.

[45] Martin Hägele, Klas Nilsson, J Norberto Pires, and Rainer Bischoff. Industrial robotics. In *Springer handbook of robotics*, pages 1385–1422. Springer, 2016.

[46] Zengxi Pan, Joseph Polden, Nathan Larkin, Stephen Van Duin, and John Norrish. Recent progress on programming methods for industrial robots. In *ISR 2010 (41st International Symposium on Robotics) and ROBOTIK 2010 (6th German Conference on Robotics)*, pages 1–8. VDE, 2010.

[47] Rainer Bischoff, Tim Guhl, Erwin Prassler, Walter Nowak, Gerhard Kraetzschmar, Herman Bruyninckx, Peter Soetens, Martin Haegele, Andreas Pott, Peter Breedveld, et al. Brics-best practice in robotics. In *Robotics (ISR), 2010 41st International Symposium on and 2010 6th German Conference on Robotics (ROBOTIK)*, pages 1–8. VDE, 2010.

[48] Erwin Prassler, Herman Bruyninckx, Klas Nilsson, and Azamat Shakhimardanov. The use of reuse for designing and manufacturing robots. *White Paper*, 2009.

[49] Byoungyoul Song, Seungwoog Jung, Choulsoo Jang, and Sunghoon Kim. An introduction to robot component model for opros (open platform for robotic services). In *Proceedings of the International Conference Simulation, Modeling Programming for Autonomous Robots Workshop*, pages 592–603, 2008.

[50] Herman Bruyninckx. Open robot control software: the orocos project. In *Proceedings 2001 ICRA. IEEE international conference on robotics and automation (Cat. No. 01CH37164)*, volume 3, pages 2523–2528. IEEE, 2001.

[51] Robot programming languages, fabryka-robotow.pl. Available at http://fabryka-robotow.pl/2015/01/programming-languages-to-control-robot (Accessed: 14th August, 2017).

[52] Cortona3D 2017. Rapid Manual, 2017. Available at http://www.cortona3d.com/rapidmanual (Accessed: 14th August, 2017).

[53] SOFTWARE KR C1 / KR C2 / KR C3 Reference Guide. Available at http://robot.zaab.org/wp-content/uploads/2014/04/KRL-Reference-Guide-v4_1.pdf (Accessed: 14th August, 2017).

[54] FANUC Robotics SYSTEM R-J3 i B Controller KAREL Reference Manual, 2003. Available at https://icdn.tradew.com/file/201606/1569362/pdf/7066348.pdf (Accessed: 19th October, 2020).

[55] ROBOTC Word Logo a C Programming Language for Robotics. Available at http://www.robotc.net/ (Accessed: 14th August, 2017).

[56] J. C Baillie, M. Nottale, and B. Pothier. The URBI Tutorial v. 1.5. Technical report, 2007.

[57] Aaron Martinez and Enrique Fernández. *Learning ROS for robotics programming*. Packt Publishing Ltd, 2013.

[58] Jean-Christophe Baillie, Akim Demaille, Quentin Hocquet, and Matthieu Nottale. Events!(reactivity in urbiscript). *arXiv preprint arXiv:1010.5694*, 2010.

[59] V. Djukić, A. Popović, and J. P. Tolvanen. Domain-Specific Modeling for Robotics: From Language Construction to Ready-made Controllers and End-user Applications. In *Proc. 3rd Work. Model. Robot Softw. Eng. (pp. 47-54). ACM.*, 2016.

[60] Claudia Pons, Gabriela Pérez, Roxana Giandini, and Gabriel Baum. Applying mda and omg robotic specification for developing robotic systems. In *International Conference on System Analysis and Modeling*, pages 51–67. Springer, 2016.

[61] Domain Level Task Force. Available at http://robotics.omg.org (Accessed: 14th August, 2017).

[62] Robotic Technology Component™ (RTC), 2012. Available at http://www.omg.org/spec/RTC/1.1 (Accessed: 14th August, 2017).

[63] David N Jansen, Holger Hermanns, and Joost-Pieter Katoen. A probabilistic extension of uml statecharts specification and verification. In *International Symposium on Formal Techniques in Real-Time and Fault-Tolerant Systems*, pages 355–374. Springer, 2002.

[64] David J Miller and R Charleene Lennox. An object-oriented environment for robot system architectures. *IEEE Control Systems Magazine*, 11(2):14–23, 1991.

[65] C Zielinski. The mrroc++ system. In *Proceedings of the First Workshop on Robot Motion and Control. RoMoCo'99 (Cat. No. 99EX353)*, pages 147–152. IEEE, 1999.

[66] Markus S Loffler, Vilas Chitrakaran, and Darren M Dawson. Design and implementation of the robotic platform. *Journal of Intelligent and Robotic Systems*, 39(1):105–129, 2004.

[67] Andreas Angerer, Alwin Hoffmann, Andreas Schierl, Michael Vistein, and Wolfgang Reif. The robotics api: An object-oriented framework for modeling industrial robotics applications. In *2010 IEEE/RSJ International Conference on Intelligent Robots and Systems*, pages 4036–4041. IEEE, 2010.

[68] Matthew O'Brien, Ronald C Arkin, Dagan Harrington, Damian Lyons, and Shu Jiang. Automatic verification of autonomous robot missions. In *International Conference on Simulation, Modeling, and Programming for Autonomous Robots*, pages 462–473. Springer, 2014.

[69] Leon Žlajpah. Simulation in robotics. *Mathematics and Computers in Simulation*, 79(4):879–897, 2008.

[70] Monica Farah-Stapleton and Mikhail Auguston. Behavioral modeling of software intensive system architectures. *Procedia Computer Science*, 20:270–276, 2013.

[71] Eric M Dashofy, André van der Hoek, and Richard N Taylor. A comprehensive approach for the development of modular software architecture description languages. *ACM Transactions on Software Engineering and Methodology (TOSEM)*, 14(2):199–245, 2005.

[72] Dave Steinberg, Frank Budinsky, Ed Merks, and Marcelo Paternostro. *EMF: eclipse modeling framework*. Pearson Education, 2008.

[73] Muhammad Usman, Aamer Nadeem, and Tai-hoon Kim. Ujector: A tool for executable code generation from uml models. In *2008 Advanced Software Engineering and Its Applications*, pages 165–170. IEEE, 2008.

[74] Erann Gat. Alfa: A language for programming reactive robotic control systems. In *Proceedings. 1991 IEEE International Conference on Robotics and Automation*, pages 1116–1117. IEEE Computer Society, 1991.

[75] Arne Nordmann and Sebastian Wrede. A Domain-Specific Language for Rich Motor Skill Architectures. In *Work. Domain-Specific Lang. Model. Robot. Syst.*, 2012.

[76] Markus Klotzbücher and Herman Bruyninckx. Coordinating robotic tasks and systems with rfsm statecharts. *JOSER: Journal of Software Engineering for Robotics*, 3(1):28–56, 2012.

[77] Gregory D Hager and John Peterson. Frob: A transformational approach to the design of robot software. In *Robotics Research*, pages 257–264. Springer, 2000.

[78] Eve Coste-Maniere and N Turro. The MAESTRO language and its environment: specification, validation and control of robotic missions. *IEEE/RSJ Int. Conf. Intell. Robot. Syst.*, pages 836—-841 vol.2, 1997.

[79] Andreas Angerer, Remi Smirra, Alwin Hoffmann, Andreas Schierl, Michael Vistein, and Wolfgang Reif. A graphical language for real-time critical robot commands. *arXiv preprint arXiv:1303.6777*, 2013.

[80] Ian Douglas Horswill. Functional programming of behavior-based systems. *Autonomous Robots*, 9(1):83–93, 2000.

[81] Alvaro Miyazawa, Pedro Ribeiro, Wei Li, Ana Cavalcanti, Jon Timmis, and Jim Woodcock. RoboChart: a State-Machine Notation for Modelling and Verification of Mobile and Autonomous Robots. *Tech. Rep.*, pages 1–18, 2016.

[82] N. Koenig and A. Howard. Design and use paradigms for gazebo, an open-source multi-robot simulator. In *2004 IEEE/RSJ Int. Conf. Intell. Robot. Syst. (IEEE Cat. No.04CH37566)*, 2004.

[83] Eric Rohmer, Surya P.N. Singh, and Marc Freese. V-REP: A versatile and scalable robot simulation framework. In *IEEE Int. Conf. Intell. Robot. Syst.*, 2013.

[84] Saadia Dhouib, Selma Kchir, Serge Stinckwich, Tewfik Ziadi, and Mikal Ziane. Robotml, a domain-specific language to design, simulate and deploy robotic applications. In *International Conference on Simulation, Modeling, and Programming for Autonomous Robots*, pages 149–160. Springer, 2012.

[85] MORSE, the Modular OpenRobots Simulation Engine. Available at https://www.openrobots.org/wiki/morse (Accessed: 14th August, 2017).

[86] MobileRobotToolkit Website. Available at http://cycabtk.gforge.inria.fr/wiki/doku.php (Accessed: 14th August, 2017).

[87] Eclipse Papyrus RobotML project a Papyrus DSML for robotic applications. Available at https://eclipse.org/papyrus/components/robotml/1.2.0/ (Accessed: 14th August, 2017).

[88] W3C. OWL Web Ontology Language, 2004.

[89] Jaime R Carbonell and Allan M Collins. Natural semantics in artificial intelligence. *American Journal of Computational Linguistics*, 1974.

[90] Anthony Mallet, Cédric Pasteur, Matthieu Herrb, Séverin Lemaignan, and Félix Ingrand. Genom3: Building middleware-independent robotic components. In *2010 IEEE International Conference on Robotics and Automation*, pages 4627–4632. IEEE, 2010.

[91] Mohammed Foughali, Félix Ingrand, and Anthony Mallet. Genom3 templates: from middleware independence to formal models synthesis. *arXiv preprint arXiv:1807.10154*, 2018.

[92] DiVA Adaptation Modelling Toolkit. Available at https://sites.google.com/site/divawebsite/divastudio/diva-adaptation-modelling-toolkit (Accessed: 14th August, 2017).

[93] Izzet Pembeci, Henrik Nilsson, and Gregory D Hager. System presentation - Functional reactive robotics: an exercise in principled integration of domain-specific languages. *Princ. Pract. Declar. Program.*, 2002.

[94] Nico Hochgeschwender, Luca Gherardi, Azamat Shakhirmardanov, Gerhard K. Kraetzschmar, Davide Brugali, and Herman Bruyninckx. A model-based approach to software deployment in robotics. *IEEE Int. Conf. Intell. Robot. Syst.*, pages 3907–3914, 2013.

[95] Xtext. Available at https://eclipse.org/Xtext/ (Accessed: 14th August, 2017).

[96] Sirius. Available at https://www.eclipse.org/sirius/ (Accessed: 14th August, 2017).

[97] pr2_navigation. Available at http://wiki.ros.org/pr2_navigation (Accessed: 14th August, 2017).

[98] pr2_pbd. Available at http://wiki.ros.org/pr2_pbd (Accessed: 14th August, 2017).

[99] RoboFlow Editor. Available at https://github.com/sonyaa/roboflow (Accessed: 14th August, 2017).

[100] Arvid Butting, Bernhard Rumpe, Christoph Schulze, Ulrike Thomas, and Andreas Wortmann. Modeling reusable, platform-independent robot assembly processes. *arXiv preprint arXiv:1601.02452*, 2016.

[101] Holger Krahn, Bernhard Rumpe, and Steven Völkel. Monticore: a framework for compositional development of domain specific languages. *International journal on software tools for technology transfer*, 12(5):353–372, 2010.

[102] Sebastian G Brunner, Franz Steinmetz, Rico Belder, and Andreas Dömel. Rafcon: A graphical tool for engineering complex, robotic tasks. In *2016 IEEE/RSJ International Conference on Intelligent Robots and Systems (IROS)*, pages 3283–3290. IEEE, 2016.

[103] RAFCON: visual state machine programming. Available at http://www.dlr.de/rmc/rm/en/desktopdefault.aspx/tabid-4845/20104_read-47199/ (Accessed: 14th August, 2017).

[104] Saddek Bensalem, Lavindra de Silva, Matthieu Gallien, Félix Ingrand, and Rongjie Yan. Rock solid" software: a verifiable and correctby-construction controller for rover and spacecraft functional levels. In *I-SAIRAS-10. Proc. of the 10th Int. Symp. on Artificial Intelligence, Robotics and Automation in Space*, 2010.

[105] Tesnim Abdellatif, Saddek Bensalem, Jacques Combaz, Lavindra De Silva, and Felix Ingrand. Rigorous design of robot software: A formal component-based approach. *Robotics and Autonomous Systems*, 60(12):1563–1578, 2012.

[106] John Peterson, Gregory D Hager, and Paul Hudak. A language for declarative robotic programming. In *Proceedings 1999 IEEE International Conference on Robotics and Automation (Cat. No. 99CH36288C)*, volume 2, pages 1144–1151. IEEE, 1999.

[107] Rajeev Alur and David L Dill. A theory of timed automata. *Theoretical computer science*, 126(2):183–235, 1994.

[108] C. A. R Hoare and He Jifeng. *Unifying Theories of Programming*. Prentice Hall College Division., 1998.

[109] Thomas Gibson-Robinson, Philip Armstrong, Alexandre Boulgakov, and A William Roscoe. Fdr3: a parallel refinement checker for csp. *International Journal on Software Tools for Technology Transfer*, 18(2):149–167, 2016.

[110] Joel Ouaknine. *Discrete analysis of continuous behaviour in real-time concurrent systems*. PhD thesis, University of Oxford, 2000.

[111] Maneesh Khattri. Translating timed automata to tock-CSP. In *Proceedings of the 10th IASTED International Conference on Software Engineering, SE 2011*, 2011.

[112] Gerd Behrmann, Alexandre David, Kim G Larsen, John Håkansson, Paul Petterson, Yi Wang, and Martijn Hendriks. UPPAAL 4.0. *Third Int. Conf. Quant. Eval. Syst. QEST 2006*, pages 125–126, 2006.

[113] Jin Song Dong, Ping Hao, Shengchao Qin, Jun Sun, and Wang Yi. Timed automata patterns. *IEEE Transactions on Software Engineering*, 34(6):844–859, 2008.

[114] Marcel Vinicius Medeiros Oliveira. *Formal derivation of state-rich reactive programs using Circus*. PhD thesis, University of York, 2005.

[115] Angela Figueiredo de Freitas. From circus to java: Implementation and verification of a translation strategy. *Master's thesis, University of York*, 2005.

[116] GHC Team. Ghc user's guide documentation, 2018.

[117] A repository for the translation of tock-CSP into Timed Automata for UPPAAL. Available at: https://github.com/ahagmj/Translation_tockCSP_TA

[118] Tom Mens and Pieter Van Gorp. A taxonomy of model transformation. *Electronic notes in theoretical computer science*, 152:125–142, 2006.

[119] Hanne Riis Nielson and Flemming Nielson. *Semantics with applications: an appetizer*. Springer Science & Business Media, 2007.

[120] Nafiseh Kahani, Mojtaba Bagherzadeh, James R Cordy, Juergen Dingel, and Daniel Varró. Survey and classification of model transformation tools. *Software & Systems Modeling*, 18(4):2361–2397, 2019.

[121] RJR Back. On correct refinement of programs. *Journal of Computer and System Sciences*, 23(1):49–68, 1981.

[122] Sidney Nogueira, Augusto Sampaio, and Alexandre Mota. Guided test generation from csp models. In *International Colloquium on Theoretical Aspects of Computing*, pages 258–273. Springer, 2008.

[123] Birgitta Lindstrom, Paul Pettersson, and Jeff Offutt. Generating trace-sets for model-based testing. In *The 18th IEEE International Symposium on Software Reliability (ISSRE'07)*, pages 171–180. IEEE, 2007.

[124] Anders Hessel, Kim G Larsen, Marius Mikucionis, Brian Nielsen, Paul Pettersson, and Arne Skou. Testing real-time systems using UPPAAL. In *Formal methods and testing*, pages 77–117. Springer, 2008.

[125] Craig Schlenoff, E Prestes, PJ Sequeira Gonçalves, Mathieu Abel, Yacine Amirat, S Balakirsky, ME Barreto, JL Carbonera, A Chibani, S Rama Fiorini, et al. Ieee standard ontologies for robotics and automation. 2015.

[126] AW Roscoe, CAR Hoare, and R Bird. The theory and practice of concurrency. 2005. *Revised edition. Only available online*.

[127] Ana Cavalcanti, Augusto Sampaio, Alvaro Miyazawa, Pedro Ribeiro, Madiel Conserva Filho, André Didier, Wei Li, and Jon Timmis. Verified simulation for robotics. *Science of Computer Programming*, 174:1–37, 2019.