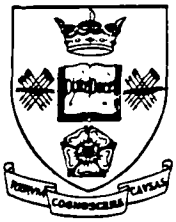
University of Sheffield



Department of Computer Science

The Integration of Software Specification, Verification, and Testing Techniques with Software Requirements and Design Processes

Wachara Chantatub

Submitted towards the degree of
Doctor of Philosophy

March 1995

To my dear grandmother

The Integration of Software Specification, Verification, and Testing Techniques with Software Requirements and Design Processes

Wachara Chantatub

Abstract

Specifying, verifying, and testing software requirements and design are very important tasks in the software development process and must be taken seriously. By investing more up-front effort in these tasks, software projects will gain the benefits of reduced maintenance costs, higher software reliability, and more user-responsive software. However, many individuals involved in these tasks still find that the techniques available for the tasks are either too difficult and far from practical or if not difficult, inadequate for the tasks.

This thesis proposes practical and capable techniques for specifying and verifying software requirements and design and for generating test requirements for acceptance and system testing.

The proposed software requirements and design specification techniques emerge from integrating three categories of software specification languages, namely an informal specification language (e.g. English), semiformal specification languages (Entity-Relationship Diagrams, Data Flow Diagrams, and Data Structure Diagrams), and a formal specification language (Z with an extended subset). The four specification languages mentioned above are used to specify both software requirements and design. Both software requirements and design of a system are defined graphically in Entity-Relationship Diagrams, Data Flow Diagrams, and Data Structure Diagrams, and defined formally in Z specifications.

The proposed software requirements and design verification techniques are a combination of informal and formal proofs. The informal proofs are applied to check the consistency of the semiformal specification and to check the consistency, correctness, and completeness of the formal specification against the semiformal specification. The formal proofs are applied to mathematically prove the consistency of the formal specification.

Finally, the proposed technique for generating test requirements for acceptance and system testing from the formal requirements specification is presented. Two sets of test requirements are generated: test requirements for testing the critical requirements, and test requirements for testing the operations of the system.

Acknowledgements

I would like to express my sincere thanks to my supervisor, Professor Mike Holcombe, who suggested the idea underlining this thesis. I am also grateful for his help, encouragement, and excellent guidance throughout my doctoral programme.

Special thanks are due to Dr. Gilbert Laycock, Dr. Colin Smythe, Dr. Matt Fairtlough, Dr. Paul McKevitt, and all members of the FormSoft, the Formal Methods and Software Engineering Group of the Computer Science Department.

I am indebted to Mr. Hugh Lafferty, Dr. Phil Green, Dr. Martin Cooke, Dr. Peter Croll, and Mr. Mark Dunn for allowing me to attend their classes.

I am grateful also to my first year examiners, Dr. Tony Cowling and Dr. John Kerridge, for their evaluation of my first year report.

Thanks to Jean Brackenbury, Karen Baker, and Gillian Callaghan, the secretarial staff at the Computer Science Department, for all their help.

This work would never been possible without the scholarship from Chulalongkorn University. I would like to express my gratitude to Professor Dr. Narasri Vivanijkul and Associate Professor Dr. Sorachai Bhisalbutra for their support. I am particularly indebted to Associate Professor Dr. Suchada Kiranandana for her endless support and encouragement.

I will not forget the kindness and hospitality of Professor Keith and Mrs. Nong Branigan and their three children, Alan, Holly, and Tanya. Thank you so much for sharing your family with me.

My thanks should also go to Mr. H. Strachey-Hawdon and Mr. and Mrs. Gittins for their kindness.

I should also thank my wonderful English teacher, Mrs. Mary Magill, for her excellent lessons and for her endless effort.

Thank you to all of my Thai friends in Sheffield for everything.

Thanks to my parents, my sister (Dr. Wacharee Attathiphaholkhun) and brothers (Mr. Wirat, Dr. Chairat, and Dr. Somrat Hirunyawasit) for their endless love and support.

My final thanks go to my husband, Thealaphan, for his patience, love, and hundreds of letters.

Contents

Chapter 1 Introduction

1.1 Overview

The objective of this chapter is to provide an overview of this thesis. This thesis proposes new techniques for specifying and verifying software requirements and design, and for generating test requirements for acceptance and system testing. Therefore, this chapter explains where these new techniques fit into the software life cycle model, why are they important, and what classes of applications will benefit from them. This chapter also lays the foundations for the rest of the thesis.

The layout of this chapter is as follows. In section 1.2, a definition of software engineering is given. Section 1.3 examines five different software life cycle models, and presents a new software life cycle model which will be followed by this thesis. It then describes phases and products within the new software life cycle model. Section 1.4 illustrates some figures to show why software requirements and design are important. Section 1.5 discusses a classification of applications and then states what classes of applications we believe will benefit from the new techniques. Section 1.6 defines and compares software requirements and design specifications. It then describes a category of software specification languages and identifies software specification languages which will be employed by the new techniques. In section 1.7, various schemes of software errors classifications are discussed and a new scheme is presented. The notions of software requirements and design verification are briefly mentioned in section 1.8. Then, in section 1.9, the notions of acceptance, system, integration, and unit test planning are briefly explained. In section 1.10, an example system, which will be used throughout this thesis, is given. Finally, in section 1.11, an overview of the rest of this thesis is noted.

1.2 Software engineering

For many years, a lot of effort has been spent in trying to find a solution to the so-called *software crisis* - software projects were being delivered far behind schedule, quality was poor, and maintenance was expensive [75]. Now the software crisis is still with us [25, 85]. Fundamentally, the software crisis is associated with the complexity of software systems coupled with the inability of techniques to deal with this complexity [21].

Software engineering is an attempt to solve the software crisis. The term *software* includes not only computer programs but also associated documentation pertaining to develop, operate, and maintain a computer system [42]. *Software engineering* is, then, defined as the application of scientific principles to the development, operation, and maintenance of computer programs and the associated documentation [7, 19, 42, 85]. Software engineering is applied in an attempt to produce *well engineered software* - software which is suitable, efficient, reliable, and maintainable, with low cost and on schedule [85].

Although there have been a lot of improvements in software engineering, various techniques and tools have been developed, the results are still far from satisfactory. Much more effort is still needed to find better solutions, better techniques and tools, to tackle the problem.

1.3 Software development process

1.3.1 Software life cycle model

In a broad sense, the software development process entails the translation of requirements into a working system which meets such requirements [92]. In order to make the software development process more manageable and more visible, the concept of the software life cycle model has come into prominence. The life cycle of a software system begins when the need for the software product is identified and ends when it is no longer used [46]. The software life cycle model comprises phases and products of those phases. A number of different software life cycle models have been proposed; some of these models can be identified as pointed out by Davis et. all [18] and Sommerville [85] and are summarized here as follows.

1) *Waterfall model*

This model views the software development process as being made up of a number of phases such as requirements analysis, design, programming, testing, operation, and maintenance. Each phase is carried out one after another, from requirements to design, from design to programming, and so on.

2) *Incremental development model*

This model views the software development process as the process of constructing a partial implementation of a total system and then later on adding increased functionality or performance until the total system is fully implemented. By following this approach, the part of the system which is already implemented can be put into operation.

3) *Prototyping model*

This model views the software development process as the process of constructing a quick partial implementation of a total system with an aim to establish the real requirements. The end-users then utilize this prototype for a period of time and supply feedback to the developers. This feedback is then used to establish the real requirements specification, which may differ from the initial requirements, then follows a re-implementation of the total system. This will ensure that the software product really meets the real requirements.

4) *Reusable model*

This model views the software development process as the process of constructing a new software system by incorporating or using components, designs, and/or programs which already exist.

5) *Automated software synthesis model*

This model views the software development process as the process of developing a formal requirements specification of a software system and transforming this specification into operational code by using correctness-preserving transformations.

The waterfall model is the most widespread model and has been used by most commercial corporations, government contractors, and governmental entities [18, 68]. Also, the waterfall model is appropriate to the software development techniques and tools available and can be adapted to specific projects [68].

In this thesis, a slight variation of the waterfall model is proposed as shown in Figure 1-1 (shaded regions represent areas of application of the techniques proposed in this thesis) and will be employed throughout this thesis. It is aimed at making explicit all important phases, products of each phase, links between phases, and links between phases and products.

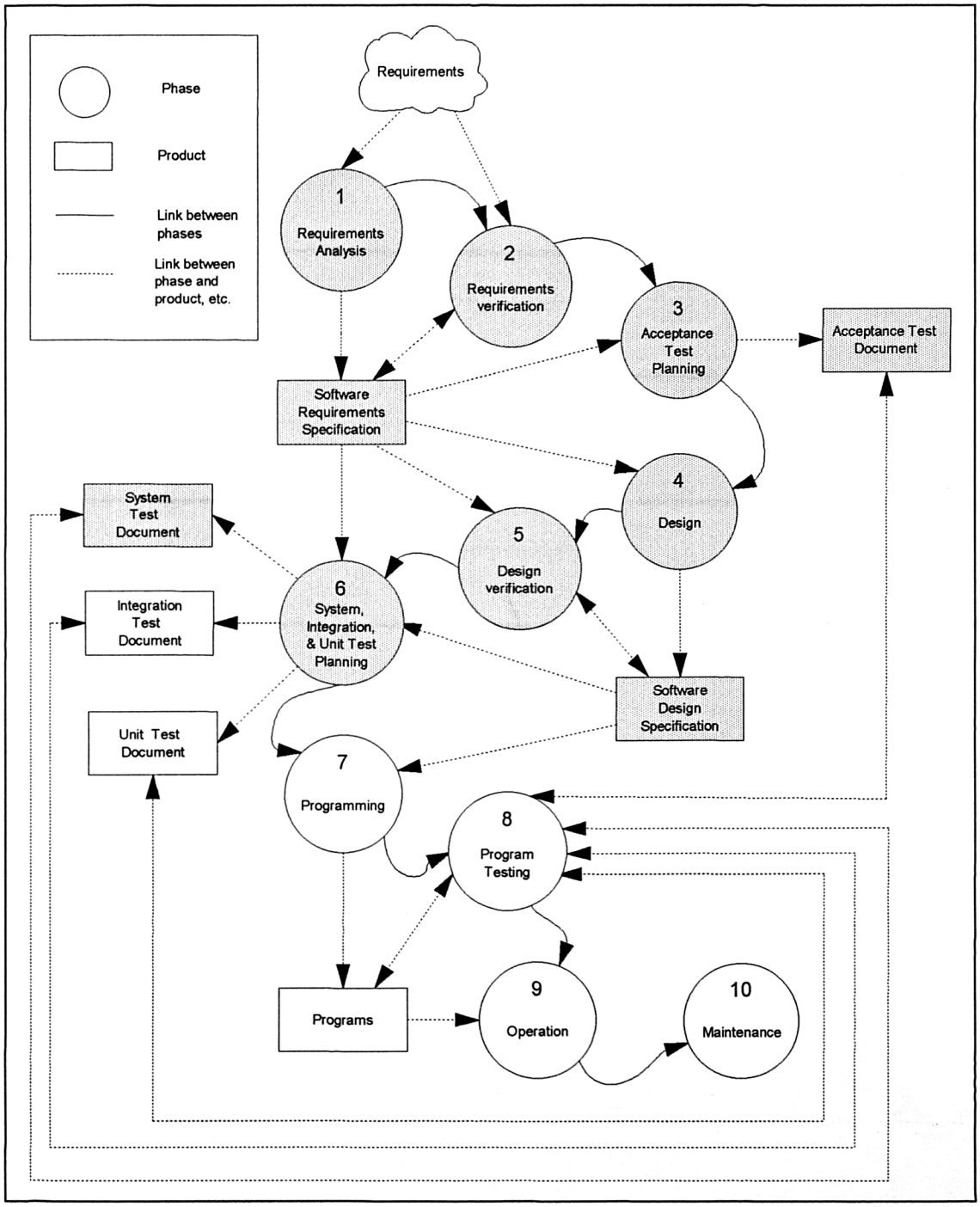


Figure 1-1: The software life cycle model (shaded regions represent areas of application of the techniques proposed in this thesis)

1.3.2 Software life cycle phases and products

According to the software life cycle model shown in Figure 1-1, there are ten phases, shown as circles, and seven products, shown as rectangles.

- The ten phases are:

  1) *Requirements analysis*

     This phase includes analyzing the end-users' requirements and producing a software requirements specification (SRS).

  2) *Requirements verification*

     This phase includes verifying the consistency within the SRS itself and verifying the consistency of the SRS against the end-users' real requirements.

  3) *Acceptance test planning*

     This phase includes producing a test plan and test specification for acceptance testing from the SRS.

  4) *Design*

     This phase includes studying the SRS, designing a software design model, and producing a software design specification (SDS).

  5) *Design verification*

     This phase includes verifying the consistency within the SDS itself, and verifying the consistency of the SDS against the SRS.

  6) *System, Integration, and unit test planning*

     This phase includes producing test plans and test specifications for system testing, integration testing, and unit testing from the SRS and SDS.

  7) *Programming*

     This phase includes transforming what has been specified in the SDS into a set of computer programs.

  8) *Program testing*

     This phase includes testing computer programs by following the unit, integration, system, and acceptance test plans and test specifications specified in the test documents.

  9) *Operation*

     This phase includes using the programs in the real practice.

  10) *Maintenance*

     This phase includes correcting errors, improving the efficiency, and adding new functions into the programs. The maintenance phase actually covers the nine phases above.

The seven products are:

1) *Software requirements specification (SRS)*

An SRS is a document containing a complete description of what the software will do in order to achieve what the end-users want without describing how it will do it [19].

2) *Acceptance test document*

An acceptance test document covers a test plan, test specification, and test report [43] for acceptance testing which is the process of testing conducted to determine whether or not a system satisfies its acceptance criteria and to enable the customer to determine whether or not to accept the system [37].

3) *Software design specification (SDS)*

An SDS is a document which records the translation of the software requirements specification into a description of the software structure, software components, interfaces, and data necessary for the programming phase [46].

4) *System test document*

A system test document covers a test plan, test specification, and test report [43] for system testing which is the process of testing an integrated hardware and software system to verify that the programs meet the specified requirements [42].

5) *Integration test document*

An integration test document covers a test plan, test specification, and test report [43] for integration testing which is the process of testing interfaces to ensure that program units are communicating as expected [37].

6) *Unit test document*

A unit test document covers a test plan, test specification, and test report [43] for unit testing which is the process of testing individual program units to ensure that the unit satisfies its functional specification [5].

7) *Programs*

Programs are computer programs produced which will be used in the operation.

This thesis concerns six phases and four products as shown with shaded regions in Figure 1-1. To be more specific, this thesis concerns: producing software requirements specifications; verifying software requirements specifications; producing software design specifications; verifying software design specifications; and generating test requirements for acceptance and system testing.

1.4 Why are software requirements and design important?

At the present, the total cost of a computer system, the cost of hardware plus the cost of software, is dominated by the cost of software. In the 1950s, approximately the cost of hardware was more than 80 percent of the total cost of a computer system, whereas the cost of software was less than 20 percent [7]. In the 1990s, the numbers are inverted; the cost of hardware is less than 20 percent of the total cost of a computer system, whereas the cost of software is more than 80 percent [84].

The cost of testing is enormous, around 50 percent of the software cost [6]. The maintenance cost is 2-4 times greater than the predelivery cost and about two-thirds of the maintenance cost can be contributed to software errors made during two early phases of software development namely the requirements analysis and design phases [55, 75]. As pointed out by Boehm [8], in large projects failure to find and correct software errors early in the life cycle can increase the cost of software by 100 times; in small projects, the number is more like 4-6 times.

In general, the later in the software life cycle that a software error is detected, the more expensive it will be to repair [19].

By studying these numbers, we then realize how important are software requirements and design. More effort needs to be spent in requirements analysis and design [93]. In other words, it is indeed very important to have correct software requirements and design specifications. Martin mentioned in [61] that "Program testing produces a crop of surprises. Integration testing produces worse surprises. Development can be even worse if the users do not like what they get. It is very expensive to deal with surprises which occur late in the cycle and which cause earlier parts of the cycle to be redone".

1.5 A classification of applications

It is necessary for developers of new techniques or tools to state explicitly what class or classes of applications will benefit from those new techniques or tools. Since this thesis proposes some new techniques, it is my obligation to state, in advance, what classes of applications would benefit from the new techniques.

There are various criteria for classifying applications, however a classification offered by Davis in [19] seems to be the most useful one and will be adapted for use in this thesis.

Applications can be classified by five important characteristics of applications as follows:

1) *Difficulty of problem*

Applications may be classified by considering the difficulty of how to solve the problem: hard (HA) or not hard (NH). An HA application is an application whose solution is unknown and it seems that it would be hard to find the solution whereas an NH application is an application whose solution is known to be applicable.

2) *Temporal relationship between input data and processes*

Applications may be classified by considering the temporal relationship between input data and processes: batch (BA) or interactive (IN). A BA application is an application whose all input data is available before the processing of the data whereas an IN application is an application whose input data is entered interactively during the processing of the data.

3) *Number of tasks performed at the same time*

Applications may be classified by considering the number of tasks to be performed at the same time: single-tasking (ST) or multi-tasking (MT). An ST application is an application which performs only one task at a time whereas an MT application is an application which is able to perform multiple tasks simultaneously.

4) *Relative difficulty of data and algorithmic aspects*

Applications may be classified by considering the complexity of data and algorithms used in solving the problem: data (DA) or algorithm (AL). A DA application is an application whose data is complex and the success of the application depends on how well the data is organized and manipulated whereas an AL application is an application in which the most difficult part of a system is how to transform the system's input into the system's output.

5) *Predictability of system output*

Applications may be classified by considering the predictability of system's outputs from a known set of inputs: deterministic (DE) or nondeterministic (ND). A DE application is an application whose outputs can be predicted in advance and the same outputs are produced, given the same inputs, whereas an ND application is an application whose outputs cannot be predicted in advance.

The techniques proposed in this thesis would be found useful in these classes of applications: NH, BA and IN, ST and MT, DA, and DE. In general, the proposed techniques can be applied to any typical business information system.

1.6 Software specifications

1.6.1 Software requirements specification vs software design specification

As shown in Figure 1-1, there are two specifications produced as the results of the requirements analysis and design phases; the two specifications are the software requirement specification (SRS) and software design specification (SDS).

As emphasized by Ward and Mellor in [92] that it is necessary to separate the SRS from the SDS and there are many benefits to be gained from the separation. Due to the fact that in practice software requirements and design processes are often not done separately and also the lack of clear agreement over whether some detail should relate to the SRS or to the SDS, the SRS and SDS sometimes do get mixed up [92]. Therefore, we often find SRSs which include design decisions. Such poorly written SRSs might limit efficient designs. The same problem is also found in SDSs; we often find SDSs which include implementation decisions. Such poorly written SDSs also might limit efficient implementations. The difficulty in distinguishing between the SRS and SDS has also been raised for example in [7, 19, 20, 85, 89].

In this section, definitions of the SRS and SDS are given, and then the benefits to be gained from the separation of the SRS from the SDS are discussed.

A *software requirements specification* (*SRS*) is a document containing a complete description of what the software will do without describing how it will do it [19]. In other words, an SRS should specify the results that must be achieved by the software, not the means of obtaining those results [44]. An SRS must correctly define all of the software requirements, but no more; in other words, it shall not describe any project management, design, implementation, or testing [44]. An SRS should be jointly prepared by both the end-user and the analyst because neither of them is qualified to prepare the SRS alone.

A *software design specification* (*SDS*) is a document which records the translation of the software requirements specification into a description of the software structure, software components, interfaces, and data necessary for the programming phase [46]. It records the

results of the design processes that are carried out during the design phase and it will be used as the medium for communicating software design information [46]. An SDS is prepared by the designer.

There are many benefits to be gained from the separation of the SRS from the SDS. First, it allows the system to be described into two different views: the SRS describes the system in terms of the subject-mattered vocabulary of the system, rather than in terms of computer hardware or software technology, so that the SRS is true regardless of the technology used to implement the system; the SDS then describes the system as actually realized by a particular technology [92]. Second, by separating the SRS from the SDS, the long-process task concerned can be broken down into two separated, but related, smaller tasks: one concerns analyzing and defining the software requirements; the other concerns designing the software and defining the software design. Finally, it permits the SRS and the SDS to be compared for consistency [92].

1.6.2 Software requirements specification (SRS)

1.6.2.1 What should and should not be included in an SRS

Concerning what should or should not be included in an SRS, some guidelines have been suggested in [19, 44] and are summarized as follows:

An SRS should include these basic issues:

1) *Functional requirements*

Functional requirements define what the software system is supposed to do; these includes the descriptions of all inputs and outputs to and from the system, the criteria for judging the validity and invalidity of all inputs and outputs, and the effects of inputs to the software system, etc.

2) *Performance requirements*

Performance requirements include the speed, response time, recovery time, etc.

3) *Attribute requirements*

Attribute requirements include portability, maintainability, security, capacity, standards compliance, etc.

4) *External interface requirements*

External interface requirements include interactions with people, hardware, other software, and other hardware, etc.

An SRS should not include these issues:

1) *Designs*

An SRS should not specify what are supposed to be design decisions such as data design and process/algorithm design.

2) *Product assurance plans*

Product assurance plans include test plans, quality assurance plans, configuration management plans, etc.

3) *Project requirements*

Project requirements include staffing, schedules, costs, milestones, activities, phases, reporting schedules, etc.

1.6.2.2 Characteristics of a good SRS

A good SRS should have the following characteristics [4, 16, 19, 44, 68]:

1) *Understandable*

An SRS is understandable if and only if everyone who uses the SRS can understand what is stated in the SRS.

2) *Unambiguous*

An SRS is unambiguous if and only if every requirement stated therein has only one interpretation.

3) *Consistent*

An SRS is consistent if and only if no subset of individual requirements stated therein conflict.

4) *Correct*

An SRS is correct if and only if every requirement stated therein represents something required of the system to be built.

5) *Complete*

An SRS is complete if and only if all requirements are stated.

6) *Verifiable*

An SRS is verifiable if and only if there exists some finite cost-effective process with which a person or machine can check that the SRS stated therein meets the real requirements.

7) *Modifiable*

An SRS is modifiable if changes to the SRS can be made easily.

8) *Traceable*

An SRS is traceable if the origin of each of its requirements is clear.

1.6.3 Software design specification (SDS)

1.6.3.1 What should and should not be included in an SDS

An SDS should include descriptions of the individual design components (design entities) and relationships among those design components [46].

The description of the individual design components, as suggested in [46], can be summarized as follows:

1) *Purpose*

Purpose is a description of why this design component exists.

2) *Function*

Function is a description of what this design component does. Function describes how this design component transforms the inputs into the desired outputs but in terms of design, not in terms of implementation.

3) *Interface*

Interface is a description of how this design component communicates with other design components.

4) *Rationale*

Rationale is a description of the factors on which the design of this component is based that are not in the SRS.

An SDS, similarly to the SRS, should not include the product assurance plans and project requirements, in addition the SDS should not include the implementations or how to implement the software system.

1.6.3.2 Characteristics of a good SDS

The characteristics of a good SDS are very similar to of a good SRS mentioned earlier. A good SDS should have the following characteristics [68, 99]:

1) *Understandable*

An SDS is understandable if and only if everyone who uses the SDS can understand what is stated in the SDS.

2) *Unambiguous*

An SDS is unambiguous if and only if every design stated therein has only one interpretation.

3) *Consistent*

An SDS is consistent if and only if no subset of individual designs stated therein conflict.

4) *Correct*

An SDS is correct if and only if every design stated therein correctly represents something stated in the SRS.

5) *Complete*

An SDS is complete if and only if the SRS is fully developed in the SDS.

6) *Verifiable*

An SDS is verifiable if and only if there exists some finite cost-effective process with which a person or machine can check that the SDS meets the SRS.

7) *Modifiable*

An SDS is modifiable if changes to the SDS can be made easily.

8) *Traceable*

An SDS is traceable if terms in the SDS have antecedents in the SRS.

1.6.4 Software specification languages

1.6.4.1 A category of software specification languages

A software specification language is the medium for communicating software requirements and design information. A variety of software specification languages is available, see [19, 69, 83,

99] for the discussion. Most of specification languages are applying to both software requirements and design specifications. Even though there are various techniques for specifying software requirements and design, the specification languages employed by those techniques are quite common.

The software specification languages may be classified into three major categories as follows:

1) *Informal specification languages*

An informal specification language is a specification language which is based on natural languages (e.g. English). Even though natural languages are considered to be user-friendly and provide more freedom of expression, they tend to be ambiguous and imprecise [91].

2) *Semiformal specification languages*

A semiformal specification language is a specification language which uses diagram (graphic) and/or semiformal textual grammar. It is generally agreed that graphical representation as well as the "English-like" nature of the semiformal specification language is much more comprehensible than just its textual counterpart [91]. Furthermore, the semiformal specification language is considered to be more precise than the informal specification language. However, the semiformal specification language is still not precise enough.

Some well-known semiformal specification languages are:

(1) *Entity-relationship diagrams* [14]

(2) *Data flow diagrams* [20, 33]

(3) *Data structure diagrams* [20, 41]

(4) *Program design language* [10]

(5) *Decision table and decision tree* [64]

(6) *Structure chart* [100]

(7) *HIPO chart* [87]

(8) *Petri nets* [70]

3) *Formal specification languages*

A formal specification language is a specification language which must have a well-defined mathematical basis; and as the result, the formal specification language is considered to be precise, can be analyzed, and mathematical proofs can be used to prove the consistency, correctness, as well as syntactic correctness of the specification [30, 97].

Some well-known formal specification languages are:

(1) *Z* [86]

(2) *VDM* [49]

(3) *OBJ* [32]

(4) *CSP* [38]

So far the SRSs and SDSs produced by particular organizations can be anything from a broad outline statement in natural language to the other extreme of a mathematically formal specification [85].

1.6.4.2 On the integration of software specification languages

Before the techniques proposed in this thesis were developed, some semiformal and formal specification languages had been explored and experimented.

- The first attempt was to try to use only one particular specification language, for example Z, VDM, and OBJ, by trying to extend the language to make it more efficient for specifying software requirements and design. However, it was found that extending the formal specification language alone does not give a satisfactory result. By integrating some semiformal specification languages with a formal specification language however the required result can be obtained.

The next step was to try to integrate different diagram techniques, for example ERDs, DFDs, DSDs, Structure chart, HIPO chart, and Petri nets, with various formal specification languages, for example Z, VDM, and OBJ. Eventually, it was found that the semiformal specification languages ERDs, DFDs, and DSDs could be integrated nicely with Z, and so it is this combination of languages that has been developed in detail in this thesis.

It is probably impractical to define a software specification by using only a single language or single category of the languages. On the other hand, it is likely that there is more to be gained by using more than one language and even more by using more than one category of the languages [3, 30, 56, 62, 88, 97]. Informal specification languages, such as English, provide more freedom of expression as well as verbal reasoning. Semiformal specification languages, such as entity-relationship diagrams, data flow diagrams, or data structure diagrams, provide for visual, reasoning, as well as having manipulation capabilities. Formal specification languages, such as Z, are precise, can be analyzed, and mathematical proofs can be used to prove the consistency, correctness, as well as syntactic correctness of the specification. However, the languages to be integrated must support each other. In addition, as pointed out by Tse in [91], the language must be convertible from one form to another so that the end-users, analysts, designers, and programmers would be able to communicate effectively.

In this thesis, new techniques for defining software requirements and design specifications are proposed. The techniques emerge from integrating the above mentioned three categories of software specification languages.

Even though the informal specification language is taking part in the new techniques, it is not discussed any further in this thesis.

Concerning the semiformal specification languages, three well-known semiformal specification languages are selected to be used in the new techniques; they are entity-relationship diagrams, data flow diagrams, and data structure diagrams. These three diagrams support one another nicely. Entity-relationship diagrams are used to model the static aspects (or state space) of the systems whereas the data flow diagrams are used to model the dynamic aspects (operations) of the systems [53]. Data structure diagrams are used to model the data structures of entities as well as the data structures of the data flows. Therefore, all aspects, except some critical requirements (data invariants) and detailed processes, of the system are captured nicely within these three diagrams.

Concerning the formal specification languages, the Z specification language [86] is selected. Z is selected because it has the facilities for separately specifying state space and operations and it also has the structuring mechanism known as schema [98].

1.7 A classification of software errors

A classification of software errors can be used in verifying, proving, or testing specifications and programs, and can be used for evaluating tools, techniques, or methodologies [15]. Software errors may be classified by symptom, by cause, by the similarity of the errors, by the software development phase in which an error was introduced, or by severity, or by some combination of these.

Various schemes of software errors classifications have been proposed, for example in [5, 15, 24, 50, 66]. In [5], software errors are classified by their similarity into the following major categories: requirements, features and functionality, structure, data, implementation and coding, integration, system and architecture, and testing. In [15], software errors are classified by a two-dimensional scheme accounting for both the software life cycle phase during which the error originated (namely requirements specifications, high level design, detailed design and coding, and additional errors incurred by altering existing products) and the behaviour or condition that caused the error (namely communicational, conceptual, and clerical). In [24], software errors are classified by the software life cycle phase during which the error originated namely requirements defects, design defects, and code defects. In [50] software errors are classified by the similarity of the errors into 13 major categories namely user interface errors, error handling, boundary-related errors, calculation errors, initial and later states, control flow errors, errors in handling or interpreting data, race conditions, load conditions, hardware, source and version control, documentation, and testing errors. In [66], software errors are classified by their cause-effect relationships.

To gain more benefit from the software errors classification, the classification should bring into attention the software development techniques being used, these include software requirements and design specification techniques, programming languages, database management system, and so on. In this thesis, a new classification of software errors is presented in which software errors are classified by the products within the software life cycle model (namely software requirements specification and software design specification) and the causes of the errors. The new classification is formulated by also bringing into attention the software requirements and design specification techniques proposed in this thesis. Since this thesis deals only with finding errors in software requirements and design specifications, the proposed classification covers only these two products as follows.

Software requirements specification

1) *Missing feature*

A feature required by the end-users but not specified in the SRS.

(1) *Missing entity or attribute*

(2) *Missing relationship*

(3) *Missing critical requirement*

(4) *Missing process*

2) *Undesirable feature*

A feature specified in the SRS but not required by the end-users.

(1) *Undesirable entity or attribute*

(2) *Undesirable relationship*

(3) *Undesirable critical requirement*

(4) *Undesirable process*

3) *Duplicate features*

A feature specified more than once in the SRS.

(1) *Duplicate entities or attributes*

(2) *Duplicate relationships*

(3) *Duplicate critical requirements*

(4) *Duplicate processes*

4) *Inconsistent features*

Features specified in the SRS are in conflict or inconsistent.

(1) *Inconsistent critical requirements*

(2) *Inconsistency between the state specification and the initial state specification*

(3) *Inconsistent interface specifications*

(4) *Inconsistency between the outputs specified in the interface specification and the outputs specified in the process specification*

(5) *Inconsistency between the process specification and the critical requirements specification*

Software design specification

Similarly to the case for the SRS, the causes of errors in the SDS are classified as follows.

1) *Missing feature*

A feature specified in the SRS but not specified in the SDS.

(1) *Missing entity or attribute*

(2) *Missing relationship*

(3) *Missing critical requirement*

(4) *Missing process*

2) *Undesirable feature*

A feature specified in the SDS but not specified in the SRS.

(1) *Undesirable entity or attribute*

(2) *Undesirable relationship*

(3) *Undesirable critical requirement*

(4) *Undesirable process*

3) *Duplicate features*

A feature specified more than once in the SDS.

(1) *Duplicate entities or attributes*

(2) *Duplicate relationships*

(3) *Duplicate critical requirements*

(4) *Duplicate processes*

4) *Inconsistent features*

Features specified in the SDS are in conflict or inconsistent.

(1) *Inconsistent critical requirements*

(2) *Inconsistency between the state specification and the initial state specification*

(3) *Inconsistent interface specifications*

(4) *Inconsistency between the outputs specified in the interface specification and the outputs specified in the process specification*

(5) *Inconsistency between the process specification and the critical requirements specification*

1.8 Requirements and design verification

It has been discovered that errors that are introduced during software requirements analysis and design phases have a major impact on the total cost of software [75]. Requirements verification and design verification are activities introduced into the software development process with an aim of finding errors introduced during the requirements analysis phase and design phase accordingly.

As shown in Figure 1-1, the requirements verification phase (generally) follows the requirements analysis phase, and the design verification phase (generally) follows the design phase. The requirements verification includes verifying the consistency within the SRS itself, and of the SRS against the end-users' requirements. The design verification includes verifying the consistency within the SDS itself, and of the SDS against the SRS.

The tasks of both requirements verification and design verification have already been studied and some general guidelines for the tasks have been presented for example in [9, 37, 45, 50, 68]. Also, many techniques have been developed in order to tackle the tasks for example [12, 39, 47, 51, 53].

1.9 Acceptance, system, integration, and unit test planning

Testing effort starts when you begin test planning. Test planning can be identified into acceptance, system, integration, and unit test planning. Each test planning is aimed to produce different test plan and test specification.

According to the ANSI/IEEE Std 829-1983 for Software Test Documentation [43], test documents cover test plans, test specifications, and test reports. Test documents may be classified into an acceptance, system, integration, and unit test document. The test plan prescribes the scope, approach, resources, and schedule or the testing activities; and it also identifies the items to be tested, the features to be tested, the testing task to be performed, the personnel responsible for each task, and the risk associated with the plan. The test specification is covered by three documents: a test-design document, test-case specification, and test procedure specification. The test report is covered by four documents: a test item transmittal, test log, test incidental, and test summary report.

There are four testing activities which are related with the four test documents mentioned above. The four testing activities are unit testing, integration testing, system testing, and acceptance testing. Unit testing is aimed to reveal that the unit does not satisfy its

functional specification. Integration testing is aimed to reveal inconsistencies among the units. System testing is aimed to reveal that the system does not satisfy the requirements. Acceptance testing is aimed to reveal that the system does not satisfy its acceptance criteria.

1.10 An example system

An example system used throughout this thesis is the library system as stated in the Fourth International Workshop on Software Specifications and Design [1] as follows.

Consider a small library database system with the following transactions:

1) Check out a copy of a book / Return a copy of a book;
2) Add a copy of a book to / Remove a copy of a book from the library;
3) Get the list of books by a particular author or in a particular subject area;
4) Find out the list of copies currently checked out by a particular borrower;
5) Find out what borrower last checked out a particular copy of a book.

There are two types of users: staff users and ordinary borrowers. Transactions 1, 2, 4, and 5 are restricted to staff users, except that ordinary borrowers can perform transaction 4 to find out the list of copies currently borrowed by themselves. The database must also satisfy the following constraints:

1) All copies in the library must be available for check out or be checked out.
2) No copy of the book may be both available and checked out at the same time.
3) A borrower may not have more than a predefined number of books checked out at one time.

This example system has been used by many authors which makes the example more interesting since it allows new authors to compare the results of the new ideas with what have already been done. This example system can be found in for example [22, 23, 27, 51, 52, 54, 57, 59, 67, 76, 77, 78, 80, 81, 82, 90, 95, 96, 102]

1.11 Overview of the thesis

The aim of this thesis is to present new techniques for *specifying* and *verifying* software requirements and design specifications, and for generating test requirements for acceptance and system testing.

In this chapter, an overview of the thesis is given.

In chapter 2, the specification languages used by the proposed software requirements and design specification techniques are discussed. Since the same set of languages are used in both specification techniques, it is appropriate to discuss them together in this chapter. The specification languages discussed are entity-relationship diagrams, data flow diagrams, data structure diagrams, and Z.

Chapter 3 presents the proposed software requirements specification technique. This chapter describes how to formulate a software requirements specification by using the specification languages explained in chapter 2.

Chapter 4 is devoted to requirements verification. This chapter explains the technique for verifying the software requirements specification produced by following the technique described in chapter 3.

Chapter 5 presents the proposed software design specification technique. This chapter describes how to develop a software design specification from the software requirements specification.

Chapter 6 is devoted to design verification. This chapter explains the technique for verifying the software design specification produced by following the technique described in chapter 5.

Chapter 7 described how to generate test requirements for acceptance and system testing from the software requirements specification.

Chapter 8 summarizes the techniques proposed in this thesis, compares them with some related works, and finally expresses the future development which may be carried out of the proposed techniques.

Chapter 2 Software specification languages

2.1 Overview

The aim of this chapter is to explain four software specification languages used in the proposed software requirements and design specification techniques: entity-relationship diagrams, data flow diagrams, data structure diagrams, and Z. These four specification languages are modified and extended and as a result they are slightly different from the ones offered by other authors. Therefore, this chapter explains their notations and briefly discusses their applications in general.

Section 2.2 describes how to draw entity-relationship diagrams (ERDs). Section 2.3 describes how to draw data flow diagrams (DFDs). Section 2.4 describes how to draw data structure diagrams (DSDs). Section 2.5 describes the extended Z subset. Finally, in section 2.6, roles of the four software specification languages in software specifications are pointed out.

2.2 Entity-relationship diagrams

An *entity-relationship diagram* (ERD) is a graphical representation of data entities (things of importance to a system about which data needs to be stored) and how they are related to one another [34]. The ERD was originally proposed by Chen [13, 14] and subsequently has been extended by many others, for example Ross [79], Flavin [29], Martin [60], and Date [17]. The ERD is used to depict the static aspects of a system. The ERD is often considered to be the most appropriate data model because it captures most of the important phenomena of the real world (entities and their relationships) and expresses them in a natural and easily understandable way [35].

There are variations of how to draw ERDs as well as the concepts captured in each variation. The way ERDs are drawn in this thesis as well as the concepts captured by them are slightly different from the others. The next section explains the notations for drawing ERDs as proposed in this thesis.

2.2.1 Notations

The notations for drawing ERDs as proposed in this thesis are:

1) Entity
2) Relationship
3) Cardinality
4) Instance participation
5) Relationship types

2.2.1.1 Entity

An *entity* is something of importance to a system about which data needs to be stored. An entity is represented by a rectangle with the name and data type of that entity inside (see Figure 2-1). In this thesis, a plural noun is used to name an entity instead of a singular noun as generally recommended. The reason for using a plural noun is that we treat an entity as a set of instances rather than an entity type. For entity types, Z data types are used.

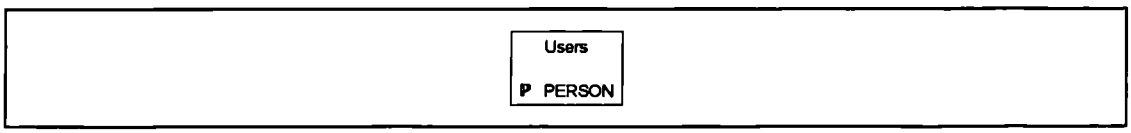


Figure 2-1: Entity

2.2.1.2 Relationship

A *relationship* is an association or link which shows how one entity or a group of entities relates to another entity or another group of entities [28]. A relationship is represented by a line connecting two entities with the name of that relationship at one end of the line. The entity which is close to the relationship name is said to be the domain of that relationship while the opposite entity is said to be the range of that relationship. We suggest using a present verb for the third singular person to name a relationship (see Figure 2-2).

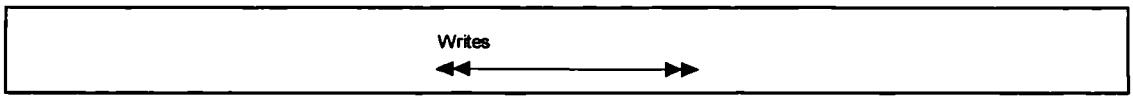


Figure 2-2: Relationship

2.2.1.3 Cardinality

In a relationship, a single instance of the entity, which shares that relationship, may participate in that relationship one or many times. The maximum number of times an instance of an entity can participate in a relationship is its *cardinality*.

The cardinality of a relationship can be either one of these:

1) *One-to-one*

A relationship whose cardinality is one-to-one relates a single instance of one entity to only a single instance of another entity. One-to-one cardinality is represented by a single-headed arrow at both ends of the relationship line (see Figure 2-3(a)).

2) *One-to-many*

A relationship whose cardinality is one-to-many relates a single instance of one entity to several instances of another entity. One-to-many cardinality is represented by a single-headed and a double-headed arrows at the ends of the relationship line accordingly (see Figure 2-3(b)).

3) *Many-to-one*

A relationship whose cardinality is many-to-one relates several instances of one entity to a single instance of another entity. Many-to-one cardinality is represented by a double-headed and single-headed arrows at the ends of the relationship line accordingly (see Figure 2-3(c)).

4) *Many-to-many*

A relationship whose cardinality is many-to-many relates several instances of one entity to several instances of another entity. Many-to-many cardinality is represented by double-headed arrows at both ends of the relationship line (see Figure 2-3(d)).

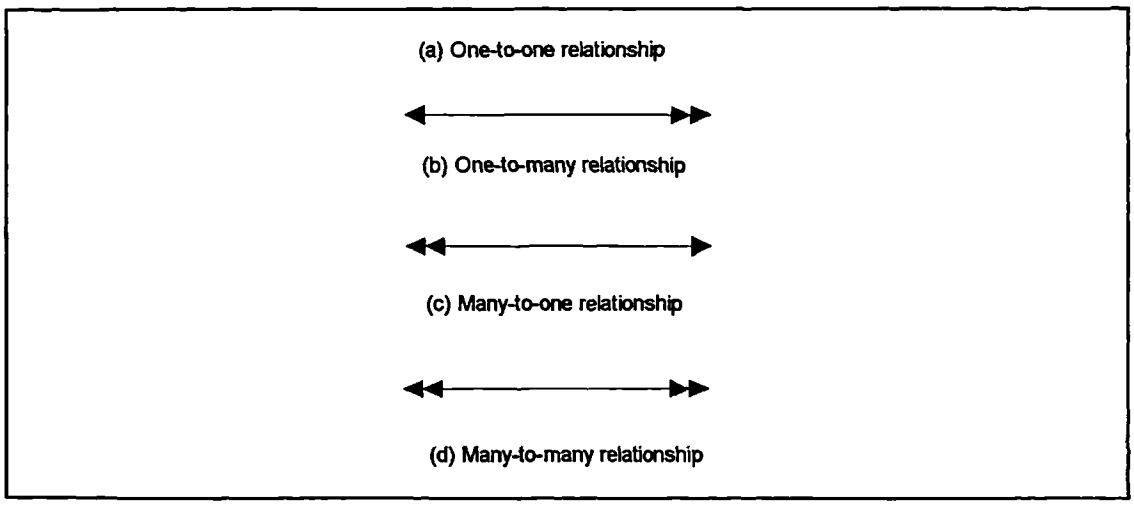


Figure 2-3: Cardinality

2.2.1.4 Instance participation

In a relationship, if every instance of the entity, which shares that relationship, must participate in that relationship, the *instance participation* of that entity is said to be *mandatory*; otherwise it is said to be *optional*. A mandatory instance participation is represented by a small solid circle (attached to the periphery of the entity box) as shown in Figure 2-4(a), and an optional instance participation is represented by a small void circle as shown in Figure 2-4(b).

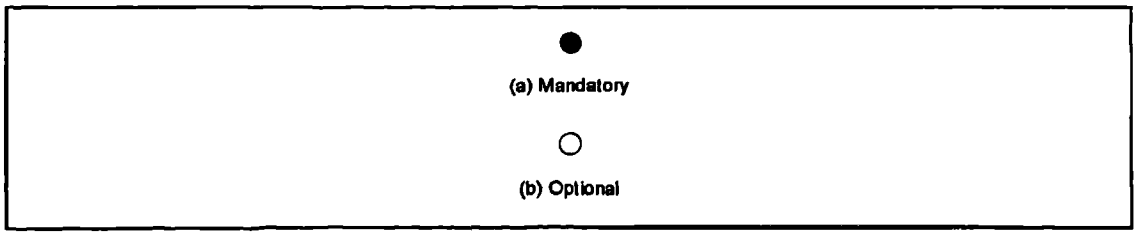


Figure 2-4: Instance participation

2.2.1.5 Relationship types

Relationships can be classified into two types: subset relationships and interrogative relationships:

1) *Subset relationships*

An entity E is a superset of entities E_1, E_2, ..., E_n, if each instance of E is also an instance of either one of the entities E_1, E_2, ..., E_n. An entity E_i is a subset of an entity E if every instance of E_i is also an instance of E. Then, a subset relationship is a relationship between the superset and its subsets. A subset relationship is a one-to-one relationship between the superset and its subsets individually and for clarity a relationship name of subset relationship is always "Is_a". Subset relationships can be further classified into:

(1) *Exclusive subset relationships*

In an exclusive subset relationship, all subsets must be mutually exclusive. The exclusive subset notation (a small circle with the letter "E" inside) as shown in Figure 2-5(a) is used to depict an exclusive subset relationship.

(2) *Inclusive subset relationships*

In an inclusive subset relationship, all subsets are not necessarily mutually exclusive. The inclusive subset notation (a small circle with the letter "I" inside) as shown in Figure 2-5(b) is used to depict an inclusive subset relationship.

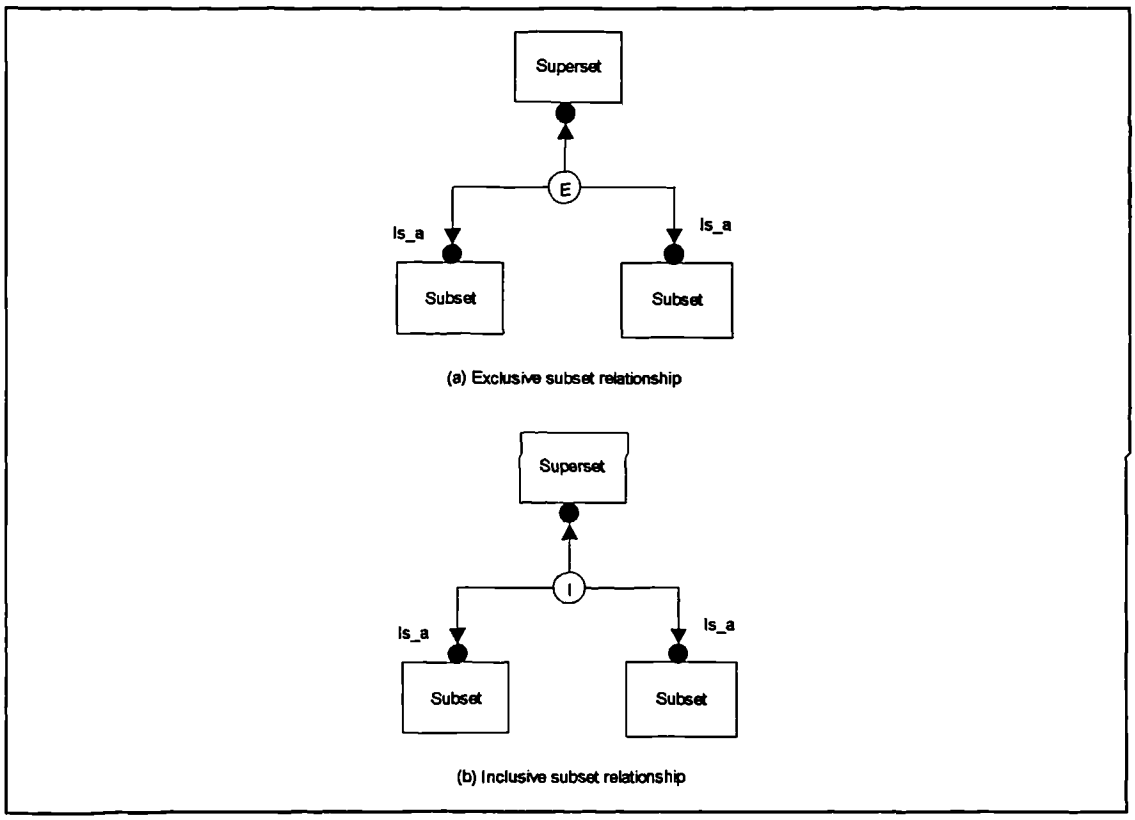


Figure 2-5: Subset relationships

2) Interrogative relationships

An interrogative relationship is a relationship (apart from a subset relationship) that needs to be kept in order to provide information required. An

interrogative relationship is drawn as shown in Figure 2-6. The relationship "Writes" in Figure 2-6 is an interrogative relationship.

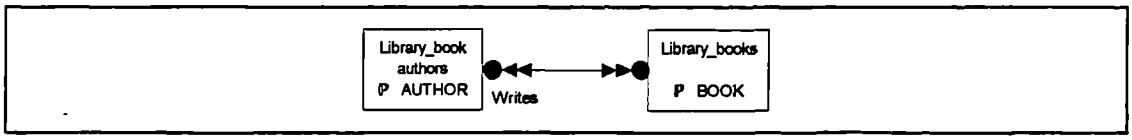


Figure 2-6: Interrogative relationship

2.2.2 Example

An example of ERDs is shown in Figure 2-7. It is the ERD of the library system (see section 1.10 for the outline of the library system). The diagram is drawn to capture the static aspects of the library system as required by the end-users and can be described as follows.

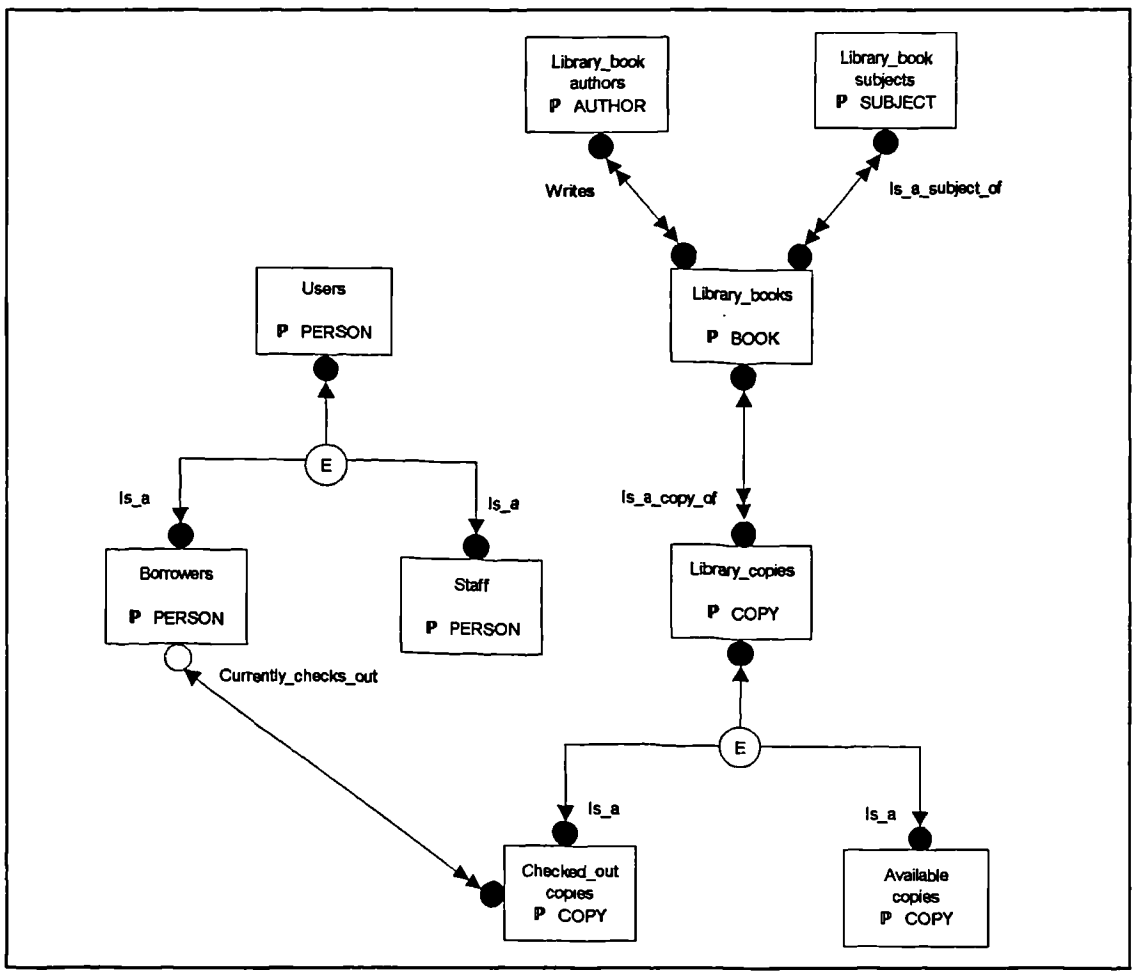


Figure 2-7: An example of ERDs

The entity "Users" is comprised of two mutually exclusive subsets "Borrowers" and "Staff" (a person cannot be both a borrower and a staff at the same time). The entity "Library_copies" is comprised of two mutually exclusive subsets "Checked_out_copies" and "Available_copies" (a library copy cannot be both a checked out copy and an available copy at the same time). As mentioned in section 2.2.1.5, the relationship name of any subset relationship is always "Is_a" and its cardinality is always one-to-one.

There is the many-to-many relationship "Writes" from the entity "Library_book_authors" to the entity "Library_books" (an author may write one or many books and a book may be written by one or many authors). The cardinality of the relationship "Writes" is mandatory-mandatory (all instances in the entities "Library_book_authors" and "Library_books" must participate in the relationship).

Also, there are the many-to-many and mandatory-mandatory relationship "Is_a_subject_of" from the entity "Library_book_subjects" to the entity "Library_books", the many-to-one and mandatory-mandatory relationship "Is_a_copy_of" from the entity "Library_copies" to the entity "Library_books", and finally the one-to-many and optional-mandatory relationship "Currently_checks_out" from the entity "Borrowers" to the entity "Checked_out_copies".

2.3 Data flow diagrams

A data flow diagram is a diagram used for specifying the movement of data through a system [83]. A DFD shows where the data flows come from, where they go to, when they leave the system, where they are stored, what processes transform them, and the interactions between data stores and the processes [34]. DFDs are used to depict the dynamic aspects of a system.

Two principal variations of how to draw DFDs are widely used: that associated with Gane and Sarson [33], and that associated with Yourdon and DeMarco [101]. However, in practice each organization normally has its own standard of how to draw DFDs. The way DFDs are drawn in this thesis is slightly different from the others, this will be explained in the next section.

2.3.1 Notations

The notations for drawing the DFDs as proposed in this thesis are:

1) External entity
2) Process
3) Data flow
4) Data store
5) Data interface
6) Data store access

2.3.1.1 External entity

External entities are sources and/or destinations of data flows. They are outside the system being developed. Data flows flow into the system only from external entities and flow out of the system only to external entities. An external entity is represented as a square with the name of that external entity inside as shown in Figure 2-8.

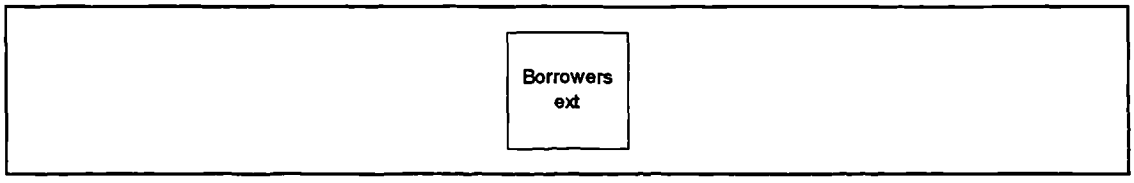


Figure 2-8: External entity

2.3.1.2 Process

Processes transform input data flows into output data flows. A process is represented as a circle with the name of that process inside. If a process is a bottom level process (no decomposition), it is represented by a thick peripheral circle (see Figure 2-9(b)), otherwise it is represented by a thin peripheral circle (see Figure 2-9(a)).

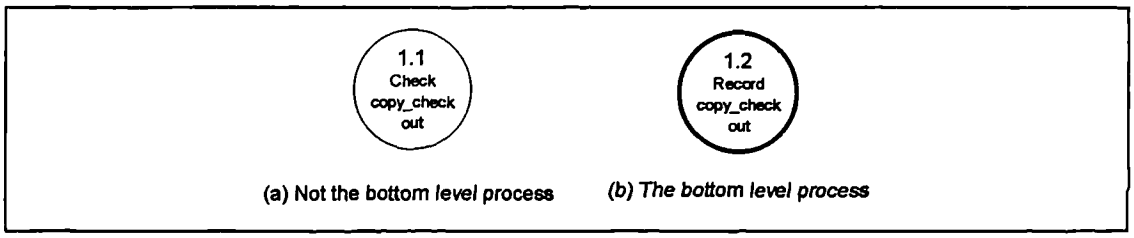


Figure 2-9: Process

2.3.1.3 Data flow

A data flow is a data item that is received or transmitted by a process. A data flow is represented as a labelled vector into or out of a process as shown in Figure 2-10. A data flow can flow between an external entity and a process, two processes, or a process and a data store. However, a data flow cannot flow between two external entities, an external entity and a data store, or two data stores. If a data flow flows between a process and a data store, a data flow label is not shown; otherwise a data flow label must be shown. A data flow can be split or decomposed into two or more data flows and a data structure diagram (DSD) is used to depict the decomposition of the data flow (see section 2.4).

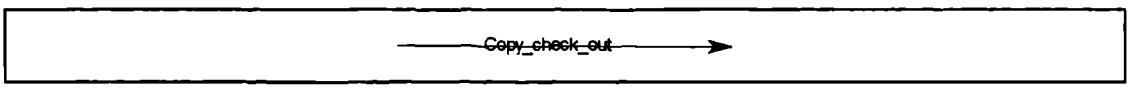


Figure 2-10: Data flow

2.3.1.4 Data store

A data store is where data is kept in order to be used by the processes. A data store is represented as a rounded rectangle with the name of that data store inside as shown in Figure 2-11. In order to keep the DFDs less messy, we recommend showing data stores only with the bottom level processes. The entities and interrogative relationships on the ERDs are the data stores on the DFDs.

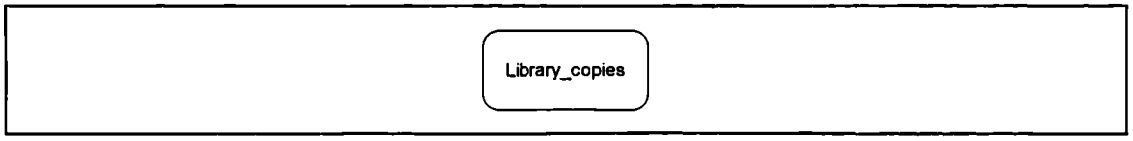


Figure 2-11: Data store

2.3.1.5 Data interface

Data interface notations are used for modelling the interfaces of input or output data flows of a process. The idea of using data interface notations are adopted from Kung in [53], but the notations and the way of drawing them as proposed in this thesis are slightly different from the one suggested in [53]. There are three data interface notations as shown in Figure 2-12. The data interface notations described in this section are closely related (in fact they capture the same concepts) to the ones described in section 2.4.1.2 and 2.5.1.3.

The three data interface notations are:

1) Conjunction

If two or more data flows are connected by a conjunction data interface, it means that all of them are required (if they are input data flows) or produced (if they are output data flows) by the process.

2) Disjunction

If two or more data flows are connected by a disjunction data interface, it means that either one or all of them are required (if they are input data flows) or produced (if they are output data flows) by the process.

3) Exclusive disjunction

If two or more data flows are connected by an exclusive disjunction data interface, it means that one and only one of them is required (if they are input data flows) or produced (if they are output data flows) by the process.

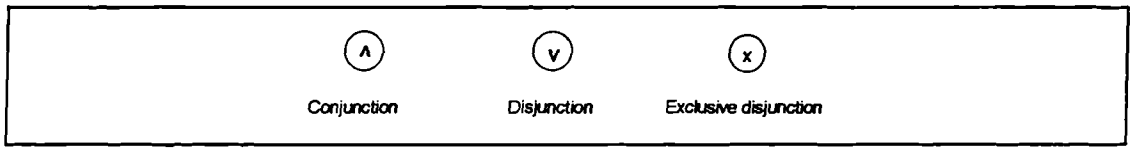


Figure 2-12: Data interface

2.3.1.6 Data store access

A data store access notation is drawn in connection with a data store notation to show what access type the process performs to the data store. There are four data store access notations as shown in Figure 2-13. If a process reads data from a data store, the read data store access notation is used and the data must flow from the process to the data store. For the remaining data store access notations, the data must flow from the process to the data store.

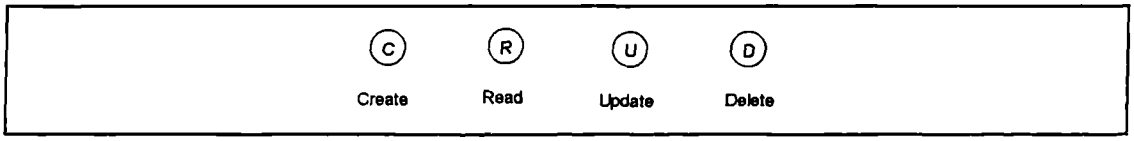


Figure 2-13: Data store access

2.3.2 Example

An example of DFDs is shown in Figure 2-14 and can be explained as follows.

There is one input data flow "Copy_check_out" flows into the process 1.1 which produces two output data flows "Invalid_copy_check_out" and "Valid_copy_check_out_check". The two output data flows are connected via the exclusive disjunction data interface notation which means that the process produces either one of the two data flows but not both.

The data flow "Valid_copy_check_out_check" along with the data flows "Borrower" and "Copy" are all required (since they are connected via the conjunction data interface notation) by the process 1.2 which produces one output data flow "Valid_copy_check_out".

The process 1.2 is the bottom level process (no decomposition), therefore it is represented by a thick peripheral circle; whereas the process 1.1 is not the bottom level process, therefore it is represented by a thin peripheral circle.

Since the process 1.2 is already the bottom level process, the data stores accessed by the process must be shown. The process 1.2 accesses three data stores: deletes an instance from the data store "Available_copies", creates an instance into the data store "Checked_out_copies", and also creates an instance into the data store "Currently_checks_out".

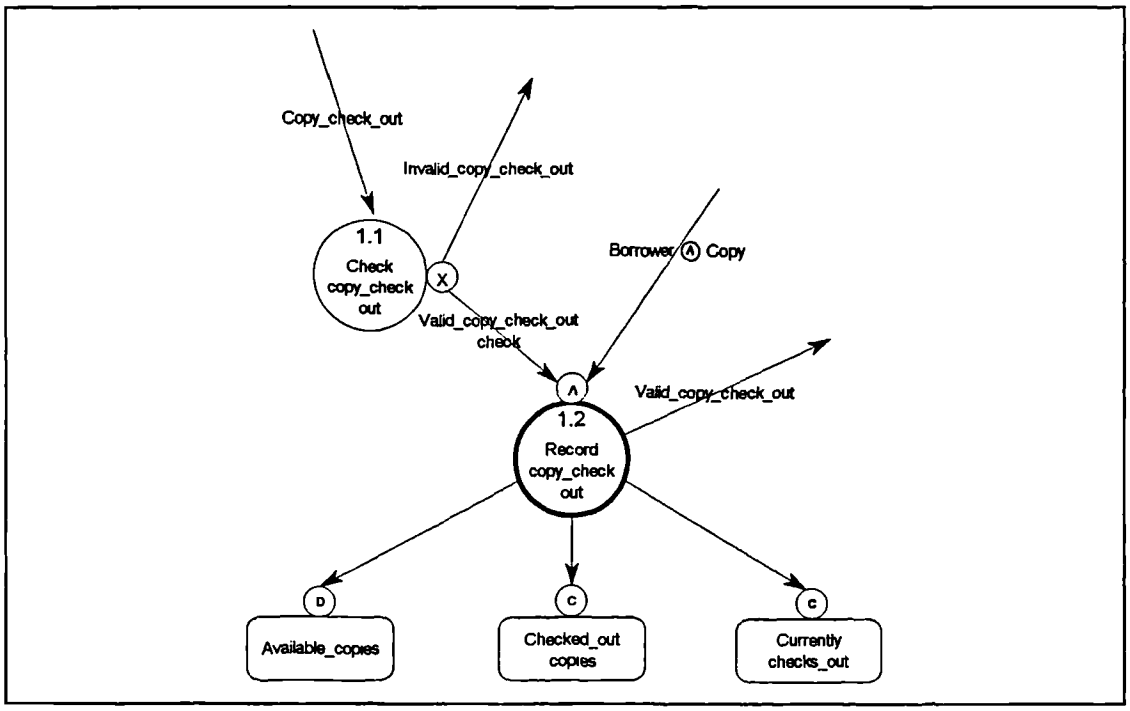


Figure 2-14: An example of DFDs

2.4 Data structure diagrams

A *data structure diagram* (DSD) is a graphical representation of the hierarchical structure of data. A data item may comprise other data items which can be depicted by a DSD. A DSD is used to show the attributes of the entity (the entity is shown in ERDs); and a DSD is also used to show the decomposition of input or output data flow (the input and output data flows are shown in DFDs).

Concerning ERDs, a DSD is drawn to show the attributes of an entity. For example, an entity "E" which comprises five attributes namely "a", "b", "c", "d", and "e". The attribute

"a" and "b" must always exist, "c" may or may not exist, and either "d" or "e" but not both must exist. A DSD can be drawn to capture the details mentioned above.

Concerning DFDs, a DSD is drawn to show the decomposition of an input or output data flow into two or more data flows. For example, a data flow "a" is decomposed into three data flows namely "b", "c", and "d", and either one or all of them may exist. A DSD can be drawn to capture these details. Since a process in DFD can be decomposed into sub-processes, a data flow can also be decomposed along with the process. Therefore, DSDs help to ensure the consistency of the data flows of the parent process and its sub-processes.

The way DSDs are drawn as proposed in this thesis is similar to the Jackson data structure diagram [20, 41]. Both capture the same concepts, but their notations are different. The notations for drawing the DSDs as proposed in this thesis are explained in the following section.

2.4.1 Notations

The notations for drawing the DSDs as proposed in this thesis are:

1) Data item

2) Data interface

3) Optional

2.4.1.1 Data item

A data item can be either an entity type, an attribute of an entity, or a data flow. A data item is represented as a rectangle with the name of that data item inside the rectangle; and if that data item is an elementary data item, its data type is shown inside the rectangle as well. Z data types are used to define the data types of data items. A data item notation is shown in Figure 2-15(a).

2.4.1.2 Data interface

There are three data interface notations as shown in Figure 2-15(b), (c), and (d). The data interface notations mentioned in this section have the same concepts as the ones described in section 2.3.1.5 and section 2.5.1.3.

2.4.1.3 Optional

If a data item may or may not exist as the component of the DSD, this can be shown by using the optional notation shown in Figure 2-15(e).

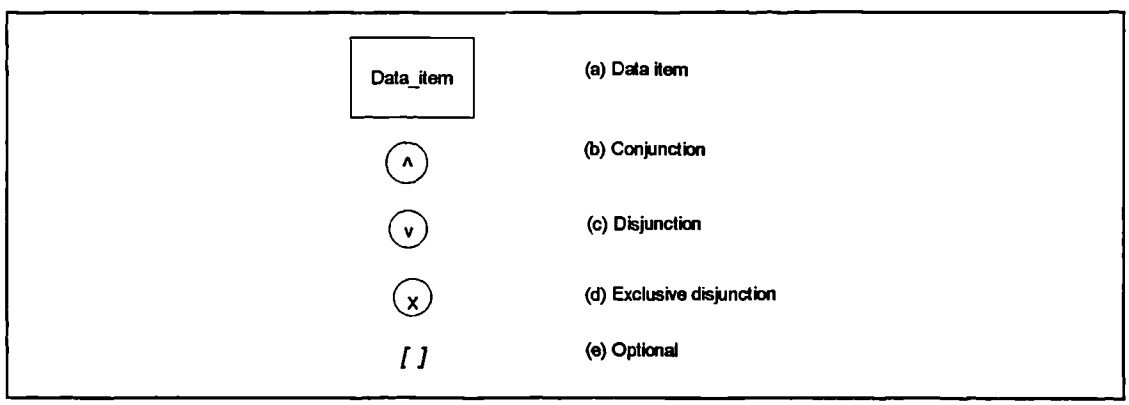


Figure 2-15: Notations for DSDs

2.4.2 Example

An example of DSDs is shown in Figure 2-16 and can be described as follows.

Figure 2-16 shows the data structure of the data item "Data_1" which is composed of the data item "Data_2", either the data item "Data_3" or the data item "Data-4" (but not both), and possibly the data item "Data_5" (since it is optional).

The data type of the data item "Data_5" is defined as a power set of the data type "TYPE5" which means that the data item "Data_5" may consist of more than one element whose type is "TYPE5", or in other words "Data_5" is a set whose element's type is "TYPE5".

If the data item "Data_1" is an entity type, this DSD shows the attributes of this entity type. If the data item "Data_1" is a data flow, this DSD shows the decomposition of this data flow.

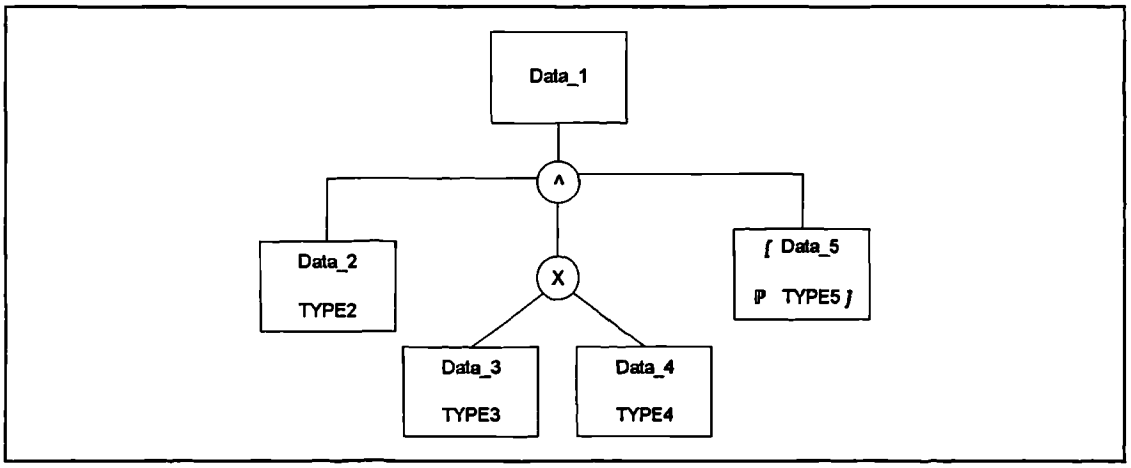


Figure 2-16: An example of the DSDs

2.5 Z

Z is a formal specification language which is based on first-order logic and set theory [22]. It is a model-based specification technique. In model-based specification techniques, the specifications of the systems that were developed modelled aspects of the system, such as its state and operations, by appropriate construction of sets, relations, functions, etc. [98]. Z was developed at Oxford University's Programming Research Group in the late seventies and early eighties. For a full account of Z the reader is referred to [86]. Currently, Z is considered as one of the most popular formal specification languages [26].

The use of Z in this thesis follows the standard given in [86]. However, some new symbols (features) for Z are necessary for the following reasons: to provide new symbols for defining important concepts which otherwise would be impossible or inappropriate if the standard Z is used; to improve the readability of Z specifications; and to provide easier and more suitable way to write Z specifications.

The extended Z subset proposed in this thesis is discussed in the following section.

2.5.1 The extended Z subset

2.5.1.1 Relationship

Relationship-decl ::= Ident1 : Ident2 Rel Ident3

Rel ::= [o<->o | •<->o | o<->• | •<->• | o<->>o | •<->>o | o<->>• | •<->>• |

o<<->o | •<<->o | o<<->• | •<<->• | o<<->>o | •<<->>o | o<<->>• | •<<->>•]

We introduce sixteen new relationship symbols as shown in Table 2-1. The new relationship symbols are used for defining not only the relation between two entities (sets) but also the cardinality and instance participation. These relationship symbols provide a more comfortable and very straight forward way for defining a relationship between two entities. They allow for a more direct connection to the ERDs described in section 2.2.

Relationship symbol	Description
1. o<->o	one-to-one, optional-optional
2. •<->o	one-to-one, mandatory-optional
3. o<->•	one-to-one, optional-mandatory
4. •<->•	one-to-one, mandatory-mandatory
5. o<->>o	one-to-many, optional-optional
6. •<->>o	one-to-many, mandatory-optional
7. o<->>•	one-to-many, optional-mandatory
8. •<->>•	one-to-many, mandatory-mandatory
9. o<<->o	many-to-one, optional-optional
10. •<<->o	many-to-one, mandatory-optional
11. o<<->•	many-to-one, optional-mandatory
12. •<<->•	many-to-one, mandatory-mandatory
13. o<<->>o	many-to-many, optional-optional
14. •<<->>o	many-to-many, mandatory-optional
15. o<<->>•	many-to-many, optional-mandatory
16. •<<->>•	many-to-many, mandatory-mandatory

Table 2-1: Relationship symbols

A relationship is a relation, the same way as a function is a relation. A function is a special case of a relation in which there is at most one instance in its range related to each instance in its domain [59]. Obviously, one-to-one and many-to-one relationships are functions. All concepts which pertain to relations are also applicable to relationships; and also all concept which pertain to functions are also applicable to one-to-one and many-to-one relationships.

The relation symbol "↔" in the standard Z and the relationship symbol "o<<->>o" in the extended Z subset are syntactically equivalent. Both represent a many-to-many and optional-optional relationship.

The partial function symbol "⇸" and the relationship symbol "o<<->o" are syntactically equivalent. Both represent a many-to-one and optional-optional relationship.

The total function symbol "→" and the relationship symbol "•<<->o" are syntactically equivalent. Both represent a many-to-one and mandatory-optional relationship.

The partial injection symbol " $\rightarrowtail\!\!\!\!\circ$ " and the relationship symbol "o$\leftrightarrow$o" are syntactically equivalent. Both represent a one-to-one and optional-optional relationship.

The total injection symbol "$\rightarrowtail$" and the relationship symbol "•$\leftrightarrow$o" are syntactically equivalent. Both represent a one-to-one and mandatory-optional relationship.

The partial surjection symbol " $\circ\!\!\!\!\twoheadrightarrow$ " and the relationship symbol "o$\leftrightarrow$•" are syntactically equivalent. Both represent a many-to-one and optional-mandatory relationship.

The total surjection symbol " $\rightarrow\!\!\!\!\twoheadrightarrow$ " and the relationship symbol "•$\leftrightarrow$•" are syntactically equivalent. Both represent a many-to-one and mandatory-mandatory relationship.

The bijection symbol " $\rightarrowtail\!\!\!\!\twoheadrightarrow$ " and the relationship symbol "•$\leftrightarrow$•" are syntactically equivalent. Both represent a one-to-one and mandatory-mandatory relationship.

There are sixteen relationship symbols and eight of them can be matched with the standard Z symbols as pointed out above. There are another eight relationship symbols which do not have the matching standard Z symbols; these eight relationship symbols are:

"o$\leftrightarrow$•"; "o$\leftrightarrow$o"; "•$\leftrightarrow$o"; "o$\leftrightarrow$•"; "•$\leftrightarrow$•";

"•$\leftrightarrow$o"; "o$\leftrightarrow$•"; "•$\leftrightarrow$•".

The relationship symbols can be defined in terms of the standard Z as follows.

1) $X \text{ o}\leftrightarrow\text{o } Y == X \rightarrowtail\!\!\!\!\circ\, Y$

2) $X \bullet\leftrightarrow\text{o } Y == X \rightarrowtail Y$

3) $X \text{ o}\leftrightarrow\bullet\ Y == \{f : X \rightarrowtail\!\!\!\!\circ\, Y \mid \text{ran } f = Y\}$

4) $X \bullet\leftrightarrow\bullet\ Y == X \rightarrow\!\!\!\!\twoheadrightarrow Y$

5) $X \text{ o}\leftrightarrow\text{o } Y == \{f : X \leftrightarrow Y \mid (\forall x_1, x_2 : X; y : Y \bullet$

$(x_1 \mapsto y) \in f \wedge (x_2 \mapsto y) \in f \Rightarrow x_1 = x_2)\}$

6) $X \bullet\leftrightarrow\text{o } Y == \{f : X \leftrightarrow Y \mid (\forall x_1, x_2 : X; y : Y \bullet$

$(x_1 \mapsto y) \in f \wedge (x_2 \mapsto y) \in f \Rightarrow x_1 = x_2) \wedge \text{dom } f = X\}$

7) $X \text{ o}\leftrightarrow\bullet\ Y == \{f : X \leftrightarrow Y \mid (\forall x_1, x_2 : X; y : Y \bullet$

$(x_1 \mapsto y) \in f \wedge (x_2 \mapsto y) \in f \Rightarrow x_1 = x_2) \wedge \text{ran } f = Y\}$

8) $X \bullet\leftrightarrow\bullet\ Y == \{f : X \leftrightarrow Y \mid (\forall x_1, x_2 : X; y : Y \bullet$

$(x_1 \mapsto y) \in f \wedge (x_2 \mapsto y) \in f \Rightarrow x_1 = x_2) \wedge \text{dom } f = X \wedge \text{ran } f = Y\}$

9) $X \text{ o}\leftrightarrow\text{o } Y == X \circ\!\!\!\!\rightarrow Y$

10) $X \bullet\leftrightarrow\text{o } Y == X \rightarrow Y$

11) $X \text{ o}\leftrightarrow\bullet\ Y == X \circ\!\!\!\!\twoheadrightarrow Y$

12) $X \bullet\leftrightarrow\bullet\ Y == X \rightarrow\!\!\!\!\twoheadrightarrow Y$

13) $X \text{ o}\leftrightarrow\text{o } Y == X \leftrightarrow Y$

14) $X \bullet\!\!\leftrightarrow\!\!\gg\!\!\circ Y == \{f : X \leftrightarrow Y \mid \mathrm{dom}\, f = X\}$

15) $X \circ\!\!\leftrightarrow\!\!\gg\!\!\bullet Y == \{f : X \leftrightarrow Y \mid \mathrm{ran}\, f = Y\}$

16) $X \bullet\!\!\leftrightarrow\!\!\gg\!\!\bullet Y == \{f : X \leftrightarrow Y \mid \mathrm{dom}\, f = X \wedge \mathrm{ran}\, f = Y\}$

Example: The relationship "Writes" (see Figure 2-7) shown in the ERD as a many-to-many and mandatory-mandatory relationship from the entity "Library_book_authors" to the entity "Library_books".

The data types of the entity "Library_book_authors" and the entity "Library_books" are defined as follows.

Library_book_authors : ℙ AUTHOR
Library_books : ℙ BOOK

The relationship "Writes" can be defined by using the relationship symbol, provided in the extended Z subset, as

Writes : Library_book_authors $\bullet\!\!\leftrightarrow\!\!\gg\!\!\bullet$ Library_books

The relationship "Writes" can also be defined in the standard Z as

Writes : AUTHOR ↔ BOOK

dom Writes = Library_book_authors

ran Writes = Library_books

The two sets of specifications are equivalent. However, the one which uses the relationship symbol in the extended Z subset seems to be more natural, shorter, and easier to understand than the one which uses the relation symbol in the standard Z.

2.5.1.2 Relationship maplet

$\mapsto$ - Relationship maplet

Given Set_x : ℙ X, and Set_y : ℙ Y, the relationship maplet notation

Set_x $\mapsto$ Set_y

is a graphical way of expressing the set of order pair (x, y) for ∀ x : Set_x; y : Set_y.

Example: Given Set_x = {1, 2, 3} and Set_y = {a, b}, then

Set_x $\mapsto$ Set_y = {(1, a), (1, b), (2, a), (2, b), (3, a), (3, b)}.

2.5.1.3 Data interface

Dt-interface ::= Ident Interface Ident /Interface Ident ..../

Interface ::= [Ⓐ | Ⓥ | Ⓧ]

The data interface symbols described here are closely related with the ones described in section 2.3.1.5 and section 2.4.1.2. The data interface symbols are introduced in order to provide a means to write Z specifications for defining data interfaces as shown in DFDs and DSDs. The same data interface notations are used as follows.

- ⓐ conjunction
- ⓥ disjunction
- ⓧ exclusive disjunction

Example: The entity type "BOOK" has four attributes namely "Id" whose type is "ID", "Isbn" whose type is "ISBN", "Issn" whose type is "ISSN", and "Title" whose type is "TITLE". The attribute "Id" and "Title" must always exist; either attribute "Isbn" or "Issn", but not both, must exist. This can be defined as a schema type as follow.

BOOK

Id : ID ⓐ (Isbn : ISBN ⓧ Issn : ISSN) ⓐ Title : TITLE

Example: The data flow "Copy_check_out" comprises three data flows namely "Requestor" whose type is "PERSON", "Borrower" whose type is "PERSON", and "Copy" whose type is "COPY"; and these three data flows must always exist. This can be defined as follow.

Copy_check_out? : (Requestor : PERSON ⓐ Borrower : PERSON ⓐ Copy : COPY)

2.5.1.4 Optional

[Ident *]* Optional

The optional symbol has the same meaning as the one described in section 2.4.1.3.

Example: The data flow "Copy_add" comprises five data flows namely "Requestor" whose type is "PERSON", "Copy" whose type is "COPY", "Book" whose type is "BOOK", "Authors" whose type is "ℙ AUTHOR", and "Subjects" whose type is "ℙ SUBJECT". All of the component data flows except "Authors" and "Subjects" must exist. Therefore the component data flows "Authors" and "Subjects" are optional (they may or may not exist). This can be defined as follow.

Copy_add? : (Requestor : PERSONⓐ Copy : COPY ⓐ Book : BOOK ⓐ
[Authors : ℙ AUTHOR*]* ⓐ *[* Subjects : ℙ SUBJECT*]*)

2.5.1.5 Input/output data flow relation

Df-relationship ::= Ident ➲ Ident

The notion that the process P produces the output data flow "O" when given the input data flow "I" is denoted by "I ➲ O", where "I" and "O" may be an elementary data flow or a data structure.

Example: If the process "R1_1_3_Check_max_copy" requires two input data flows, "Borrower" and "Valid_borrower_copy_check_out", in order to produce either the output data flow "Within_limit" or "Over_limit" but not both. This can be defined as follow.

Borrower? ⓐ Valid_borrower_copy_check_out? ➲ Within_limit! ⊗ Over_limit!

2.5.1.6 Data flow passing

External-passing ::= ⇦ Ident1 (Ident,..., Ident)

Parent-passing ::= ⇧ Ident1 (Ident,..., Ident)

Sibling-passing ::= ⇨ Ident1 (Ident,..., Ident)

A data flow can be passed between: 1) an external entity and a process; 2) a parent process and its sub-processes; 3) two sibling processes or in other words between two processes within the same level.

Ident1 is the schema name which the Ident belongs to. Ident is the variable defined (as either an input or output variable) in the schema Ident1.

Example: In the Z schema "R1_2_Record_copy_check_out", there are two declaration statements as follows.

⇧ R1_Check_out_copy (Borrower?, Copy?, Valid_copy_check_out!)

⇨ R1_1_Check_copy_check_out (Valid_copy_check_out_check?)

The first statement states that its parent schema, the schema "R1_Check_out_copy", passes two input data flows "Borrower" and "Copy" and will receive one output data flow "Valid_copy_check_out" from the schema being defined, the schema "R1_2_Record_copy_check_out".

The second statement states that its sibling schema, the schema "R1_1_Check_copy_check_out", passes one input data flow "Valid_copy_check_out" and will not receive any output data flow from the schema being defined.

2.6 Roles of software specification languages in software specifications

The four specification languages described above, ERDs, DFDs, DSDs, and Z with the extended subset, will be used for defining both *software requirements and design specifications.*

According to the software requirements and design specification techniques proposed in this thesis, software requirements specifications and software design specifications are composed of informal text, ERDs, DFDs, DSDs, and Z specifications. These specification languages are integrated in order to produce a software requirements specification or software design specification as a whole.

Even though, I strongly recommend including informal text in both SRSs and SDSs, it is beyond the scope of this thesis.

When the four specification languages are used for defining software requirements specifications, they are drawn or written to capture the requirements as required by the end-users: and we will call the diagrams and Z specifications produced as follows: Required Entity-Relationship Diagrams (RERDs); Required Data Flow Diagrams (RDFDs); Required Data Structure Diagrams (RDSDs); and Required Z specifications (RZs). Chapter 3 shows how to define software requirements specifications by using these four specification languages.

When they are used for defining software design specifications, they are drawn or written to capture the designer's design decisions; and we will call the diagrams and Z specifications produced as follows: Designed Entity-Relationship Diagrams (DERDs); Designed Data Flow Diagrams (DDFDs); Designed Data Structure Diagrams (DDSDs); and Designed Z specifications (DZs). Chapter 5 shows how to define software design specifications by using these four specification languages.

Chapter 3 Software requirements specification technique

3.1 Overview

The objective of this chapter is to describe the software requirements specification technique proposed in this thesis. The proposed SRS technique will be explained step by step and the technique is illustrated using the example system given in section 1.10, the library system.

Section 3.2 gives an overview of the proposed SRS technique: the static and dynamic aspects of a software system is discussed, and the steps of the proposed SRS technique are given. Section 3.3 describes how to draw ERDs and DSDs to define the static aspects of a system as required by the end-users. Section 3.4 describes how to draw DFDs and DSDs to define the dynamic aspects of a system as required by the end-users. Finally, section 3.5 describes how to write Z specifications to define formally both the static and dynamic aspects of a system.

3.2 Overview of the proposed SRS technique

As shown in the software development life cycle model given in Figure 1-1 (section 1.3.1), a software requirements specification (SRS) is a product of the requirements analysis phase. There are two distinct tasks in the requirements analysis phase: problem analysis and product description. The problem analysis is the task of studying end-users' needs and finding all the constraints on those requirements. The product description is the task of producing the software requirements specification.

The proposed SRS technique uses four specification languages: ERDs, DFDs, DSDs, and Z. These four specification languages are modified and extended (as explained in chapter 2) so that they can support each other nicely and are capable of producing good SRSs (as described in section 1.6.2).

As mentioned in section 2.6, we call the ERDs, DFDs, DSDs, and Z specifications which are produced to specify software requirements the RERDs (Required Entity-Relationship Diagrams), RDFDs (Required Data Flow Diagrams), RDSDs (Required Data Structure Diagrams), and RZs (Required Z specifications) accordingly.

3.2.1 The static and dynamic aspects of a system

One way to describe a software system is to describe it in terms of both static and dynamic aspects of that system. The *static aspects* of a system include: the state the system can occupy; and the critical requirements or data invariants that must hold in every state. The *dynamic aspects* of a system include the processes or operations that change the state and that produce the required information from the state.

3.2.1.1 The static aspects of a system

Following the proposed SRS technique, the static aspects of a system are specified graphically by using entity-relationship diagrams (ERDs) and, possibly, by using data structure diagrams (DSDs); and they are also specified formally by using Z specifications.

RERDs are drawn to depict entities and their relationships which are important to the system. In other words, RERDs are drawn to depict the state the system can occupy and also the critical requirements that must hold. However, there might be some critical requirements which cannot be shown on the RERDs.

RDSDs are drawn to describe the attributes of the entities. RDSDs may not be drawn if the specifiers decide not to describe the attributes of the entities in the SRS.

The static aspects of a system are also specified formally in RZs. The RZs define both the state and the critical requirements of the system. Some critical requirements which cannot be shown on RERDs must be fully specified in the RZs.

Therefore, by using the RERDs, RDSDs and RZs to define the static aspects of a system, we achieve the specifications of the static aspects of a system which, we believe, are both easy to understand and precise.

3.2.1.2 The dynamic aspects of a system

Following the proposed SRS technique, the dynamic aspects of a system are specified graphically by using data flow diagrams (DFDs) and data structure diagrams (DSDs); and they are also specified formally by using Z specifications.

RDFDs are drawn to describe processes (operations) that change the state of the system and that acquire information from the state. However, the RDFDs can only define the decomposition of the processes and the relationship among the inputs or outputs of each process, but cannot define the detailed operations of the processes.

RDSDs are drawn to define the data components or the decomposition of the input or output data flows. An input or output data flow may comprise other data components (other data flows), and an RDSD is drawn to depict its data structure. Since data flows can be decomposed along with the processes, RDSDs are used to ensure the consistency of the input and output data flows of the parent and its sub-processes.

The dynamic aspects of a system are also specified formally in RZs. The RZs define the complete dynamic aspects of a system: the decomposition of the processes, the relationship among the inputs or outputs of each process, and the detailed operations of the processes.

Therefore, by using the RDFDs, RDSDs, and RZs to define the dynamic aspects of a system, we achieve the specifications of the dynamic aspects of a system which, we believe, are both easy to understand and precise.

3.2.2 Steps of the proposed SRS technique

The proposed SRS technique is applied using the following steps:

1) Draw RERDs and RDSDs
    1.1) Identify all entities and their relationships
    1.2) Draw RERDs
    1.3) Draw RDSDs

2) Draw RDFDs and RDSDs
- 2.1) Draw the context RDFD and RDSDs
  - 2.1.1) Identify all external entities and input and output data flows
  - 2.1.2) Draw the context RDFD
  - 2.1.3) Identify the data components of each input and output data flow
  - 2.1.4) Draw RDSDs
- 2.2) Draw the next level RDFDs and RDSDs
  - 2.2.1) Identify sub-processes
  - 2.2.2) Identify input and output data flows of each sub-process
  - 2.3.3) Draw the next level RDFDs
  - 2.2.4) Identify the data components of each new internal data flow
  - 2.3.5) Draw RDSDs

3) Write RZs
- 3.1) Define the state of the system
- 3.2) Define the initial state of the system
- 3.3) Define the operations of the system

In the following sections, each step given above will be described in detail and the library system will be used as an example system to illustrate the technique.

3.3 Step 1: draw RERDs and RDSDs

3.3.1 Step 1.1: identify all entities and their relationships

To draw RERDs, first all entities as well as their relationships must be identified (see [101] for a guideline of how to identify entities and their relationships). Then, each entity is assigned a data type. Finally, the relationships among those entities are identified.

According to the library system described in section 1.10, the following entities are identified: "Users" (all users of the library), "Borrowers" (all borrowers of the library), "Staff" (all staff of the library), "Library_book_authors" (authors of all library books), "Library_book_subjects" (subjects of all library books), "Library_books" (all books in the library), "Library_copies" (copies of all books in the library), "Checked_out_copies" (all checked out copies), and "Available_copies" (all copies available to be checked out).

Next, each entity is assigned a data type. The Z data types are used. The entities "Users", "Borrowers", and "Staff" are assigned the same data type as "ℙ PERSON". The entity "Library_book_authors" is assigned the data type as "ℙ AUTHOR". The entity "Library_book_subjects" is assigned the data type as "ℙ SUBJECT". The entity "Library_books" is assigned the data type as "ℙ BOOK". The entities "Library_copies", "Checked_out_copies", and "Available_copies" are assigned the same data type as "ℙ COPY".

Then the relationships of those entities are identified as follows. The entities "Borrowers" and "Staff" are subsets of the entity "Users", and the instances (members/elements) of the two subsets are mutually exclusive. Similarly, the entities "Checked_out_copies" and "Available_copies" are mutually exclusive subsets of the entity "Library_copies". There are four interrogative relationships: "Writes", "Is_a_subject_of", "Is_a_copy_of", and "Currently_checks_out". "Writes" is a many-to-many and mandatory-mandatory relationship from "Library_book_authors" to "Library_books". "Is_a_subject_of" is a many-to-many and mandatory-mandatory relationship from "Library_book_subjects" to "Library_books". "Is_a_copy_of" is a one-to-many and mandatory-mandatory relationship

from "Library_books" to "Library_copies". Finally, "Currently_checks_out" is a one-to-many and optional-mandatory relationship from "Borrowers" to "Checked_out_copies".

3.3.2 Step 1.2: draw RERDs

From the entities and their relationships identified in step 1.1, RERDs of the system can be drawn. The RERDs are drawn by following the notations given in section 2.2.1.

The RERD of the library system is shown in Figure 3-1.

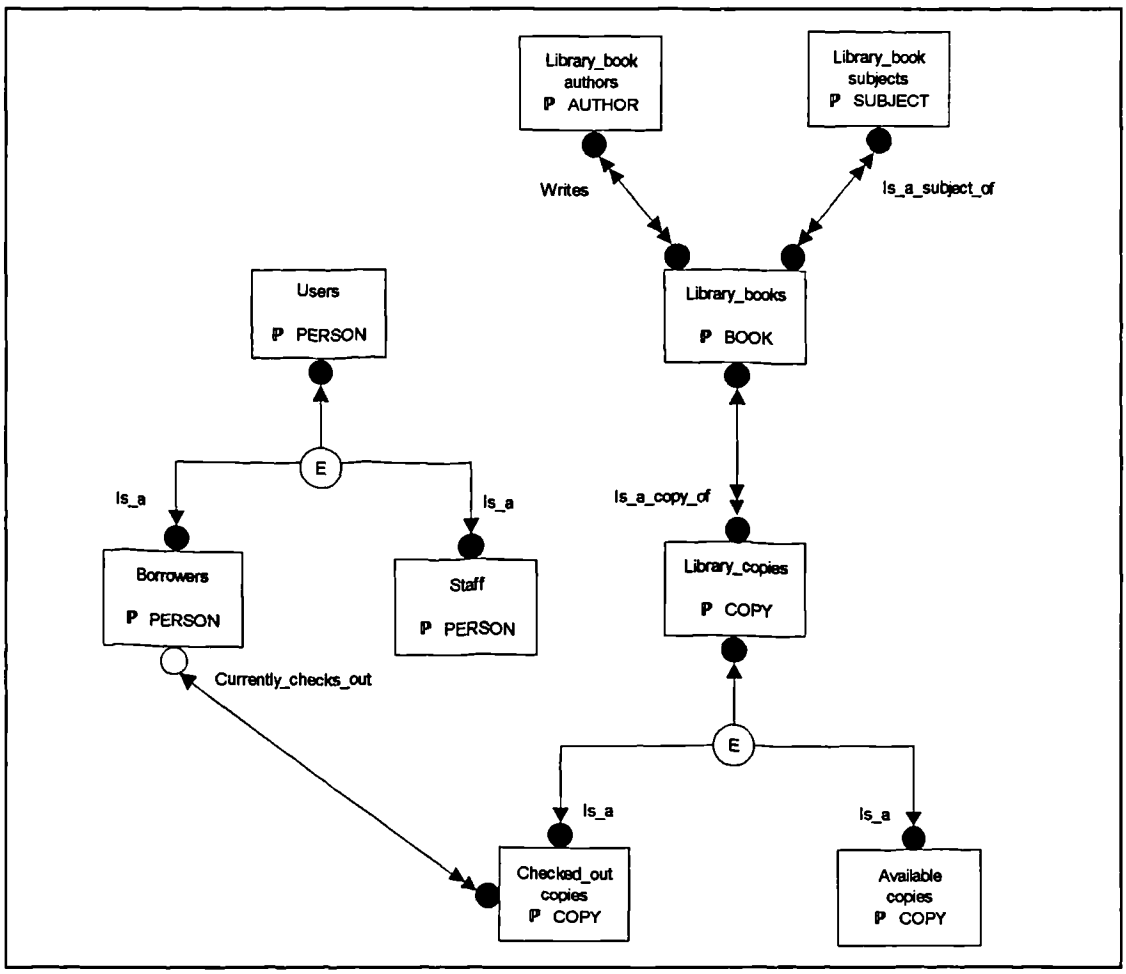


Figure 3-1: The RERD of the library system

3.3.3 Step 1.3: draw RDSDs

If the specifiers decide to define the attributes of some or all entity types identified in the step 1.1, the attributes of those entity types must be identified. Then, each attribute is assigned a data type.

If the attributes of the entity types were identified, RDSDs are drawn, by following the notations given in section 2.4.1, to depict the data structure of those entity types.

However, for the library system, we assume that the specifiers have decided not to define the attributes of the entity types. Therefore, no RDSDs are drawn.

3.4 Step 2: draw RDFDs and RDSDs

3.4.1 Step 2.1: draw the context RDFD and RDSDs

3.4.1.1 Step 2.1.1: identify all external entities and input and output data flows

To draw the context RDFD, first all external entities and input and output data flows from/to those entities must be identified. Then, the interfaces of the input as well as output data flows must be identified.

Concerning the library system, there are two external entities: "Borrowers_ext" (all borrowers of the library) and "Staff_ext" (all staff of the library).

The input data flows from the "Borrowers_ext" can be identified as follows: "Copy_check_out", "Copy_return", "By_author_enquiry", "By_subject_enquiry", and "By_borrower_enquiry".

The output data flows to the external entity "Borrowers_ext" are as follows: "Valid_copy_check_out", "Invalid_copy_check_out", "Valid_copy_return", "Invalid_copy_return", "List_of_books_by_author", "Invalid_by_author_enquiry", "List_of_books_by_subject", "Invalid_by_subject_enquiry", "List_of_copies_by_borrower", "Invalid_by_borrower_enquiry".

Similarly, the input and output data flows from/to the external entity "Staff_ext" can also be identified.

Then, the interfaces of the input and the output data flows must be identified. For example, the system either produces the output data flow "Valid_copy_check_out" or "Invalid_copy_check_out" but not both. Therefore, these two output data flows are interfaced via an exclusive disjunction notation.

3.4.1.2 Step 2.1.2: draw the context RDFD

Then, the context RDFD of the system is drawn.

The context RDFD of the library system is shown in Figure 3-2.

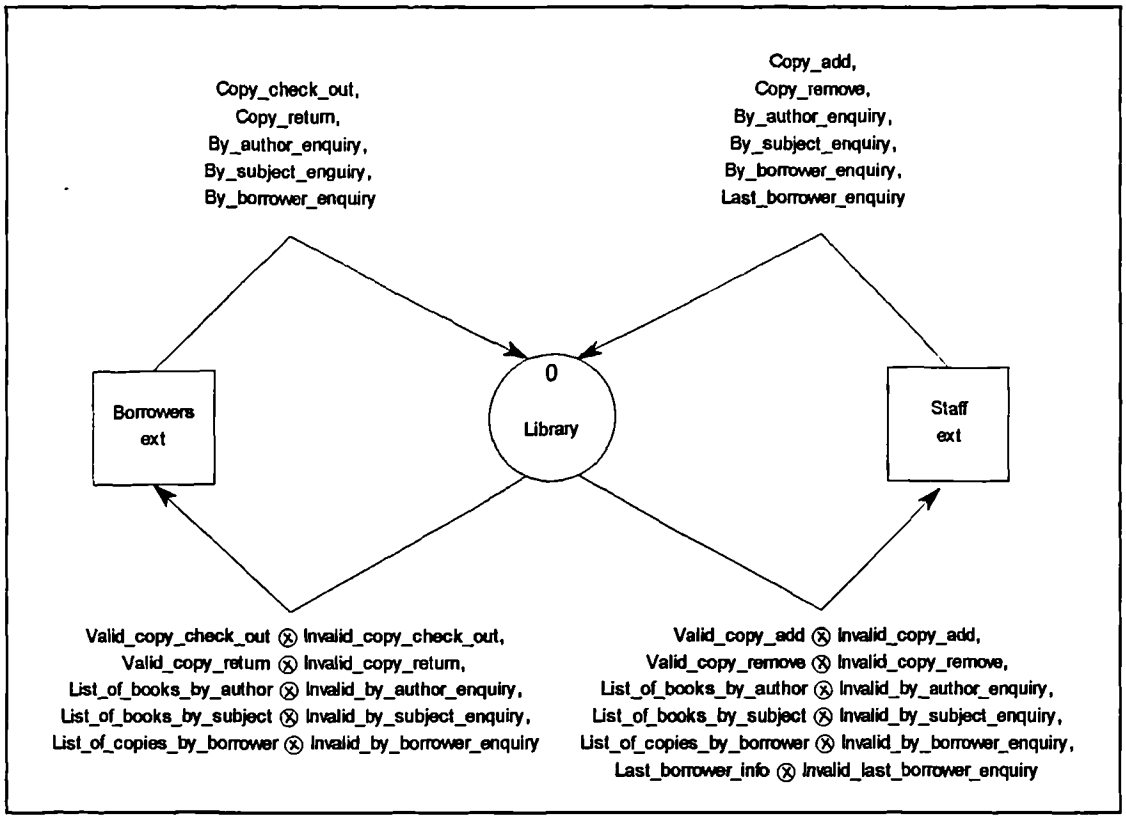


Figure 3-2: The context RDFD of the library system

3.4.1.3 Step 2.1.3: identify the data components of each input and output data flow

Next, the data components of each input and output data flow must be identified as well as the interfaces of those data components. Then each data component is assigned a data type.

The input data flow "Copy_check_out" comprises three data components (data flows): "Requestor", "Borrower", and "Copy". In addition, these three data components must be exist, therefore "Copy_check_out" is a conjunction of these three data components.

The output data flow "Valid_copy_check_out" has no data component.

The output data flow "Invalid_copy_check_out" comprises four data components (data flows): "Invalid_requestor", "Invalid_borrower", "Over_limit", and "Invalid_copy". In addition, either one of these output data flows or a combination of them can be produced: "Invalid_requestor", either "Invalid_borrower" or "Over_limit" but not both, or "Invalid_copy". Therefore the data components "Invalid_borrower" and "Over_limit" are in exclusive disjunction, and these two data components are combined with the data components "Invalid_requestor" and "Invalid_copy" using disjunction.

The data components of other input and output data flows can be identified in a similar way.

Then each data component is assigned a data type.

3.4.1.4 Step 2.1.4: draw RDSDs

Next, a RDSD is drawn to depict the data structure of each input and output data flow.

The RDSDs of the input and output data flows on the context RDFD are shown in Figures 3-3 to 3-26.

Some of the RDSDs will be explained as follows; other RDSDs can be explained in a similar way.

From Figure 3-3, the data flow "Copy_check_out" is defined as the conjunction of three data flows namely "Requestor", "Borrower", and "Copy". Therefore, the three data flows must exist. This can be explained as follows. When a borrower wants to borrow a copy of a book from the library, the input data flow "Check_out_copy" is sent into the system. The data that are required in order that the system would be able to process the transaction are "Requestor" (who operates the transaction), "Borrower" (who borrows the copy) and "Copy" (which copy is to be borrowed).

From Figure 3-5, the data components "Authors" and "Subjects" are optional (they may or may not exist). If a copy is a copy of a new book, the two data components are required, otherwise they are not required. The data type of the data component "Authors" is "ℙ AUTHOR" because a book may be written by one or many authors. The data type of the data component "Subjects" is "ℙ SUBJECT" because a book may be classified into one or many subjects.

From Figure 3-11, the data flow "Valid_copy_check_out" has no data components because only one message is produced to inform the operator that the copy is successfully checked out.

From Figure 3-12, the data flow "Invalid_copy_check_out" has four data components: "Invalid_requestor", "Invalid_borrower", "Over_limit", and "Invalid_copy". The data components are in disjunction and exclusive disjunction. This can be interpreted as follow. The copy cannot be checked out if either one or the combination of these cases happened: the requestor is invalid, the borrower is invalid, or the borrower is valid but the borrower has borrowed the maximum number of copies already, or the copy is invalid.

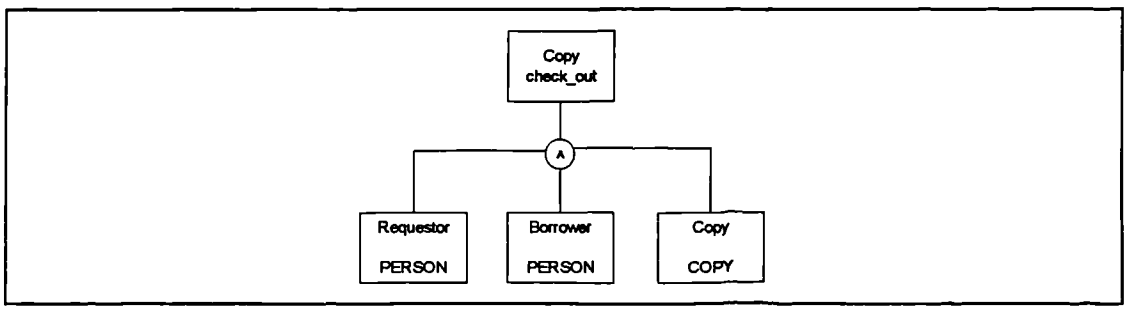


Figure 3-3: The RDSD of the data flow "Copy_check_out"

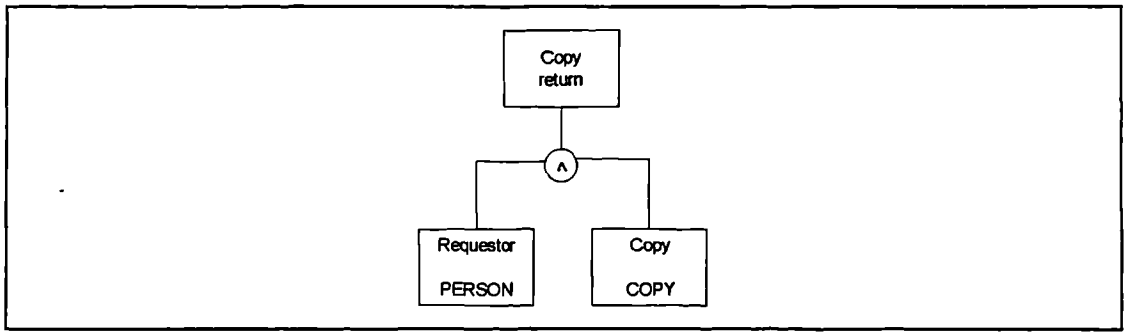


Figure 3-4: The RDSD of the data flow "Copy_return"

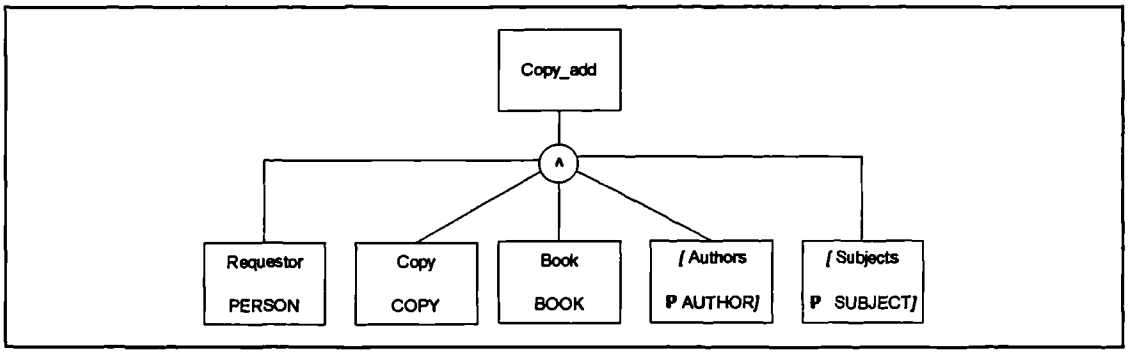


Figure 3-5: The RDSD of the data flow "Copy_add"

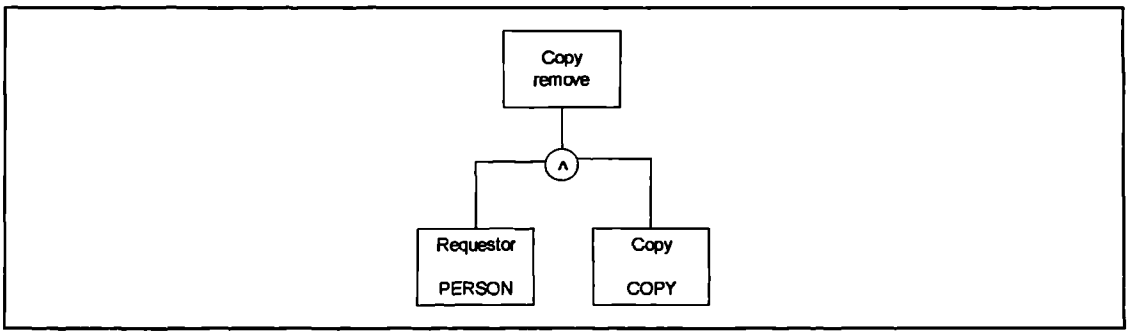


Figure 3-6: The RDSD of the data flow "Copy_remove"

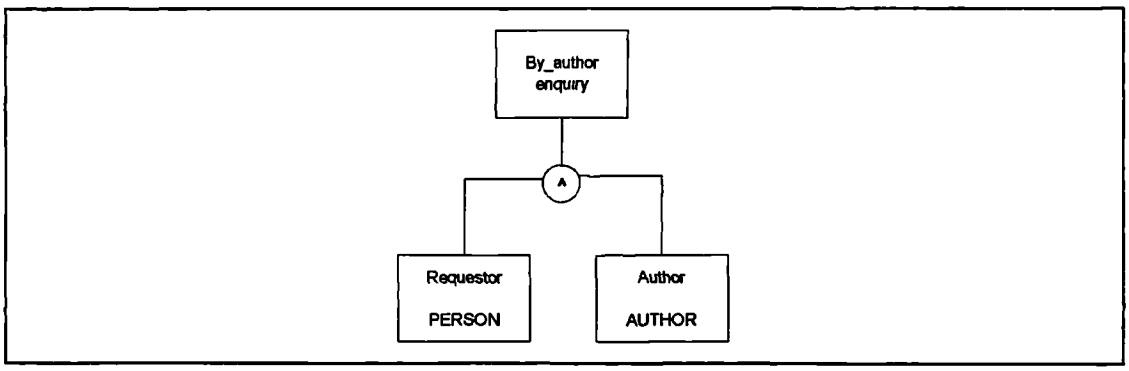


Figure 3-7: The RDSD of the data flow "By_author_enquiry"

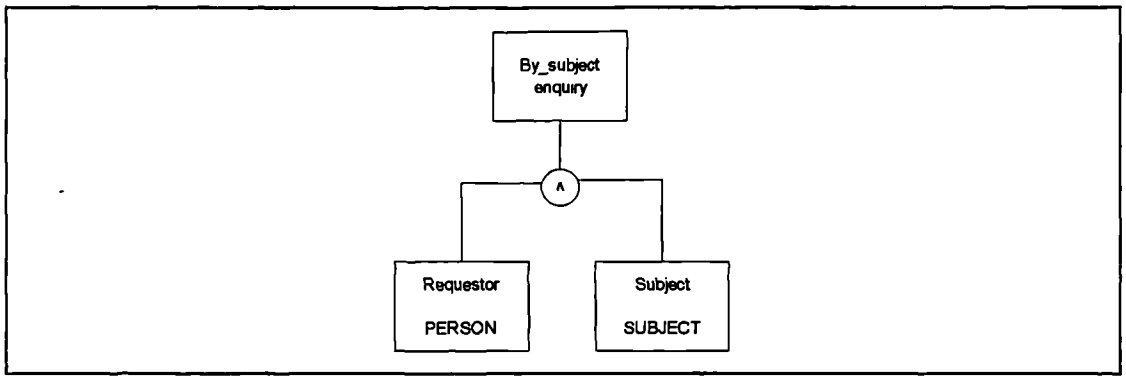


Figure 3-8: The RDSD of the data flow "By_subject_enquiry"

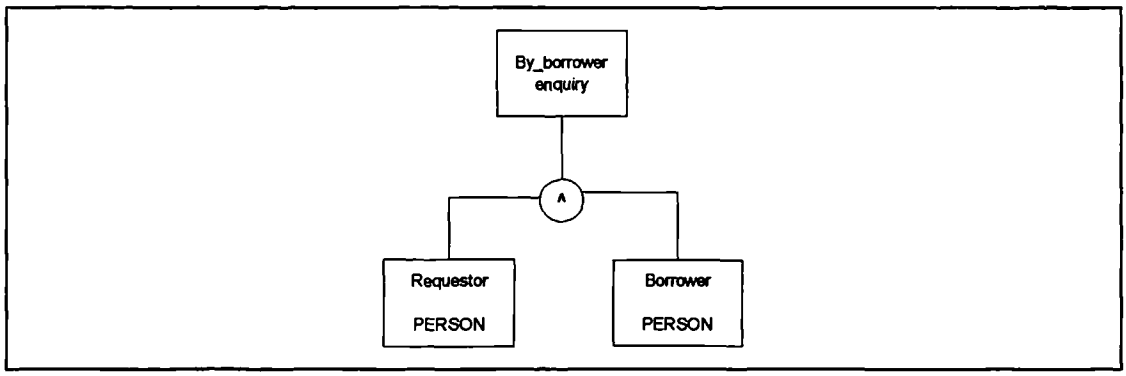


Figure 3-9: The RDSD of the data flow "By_borrower_enquiry"

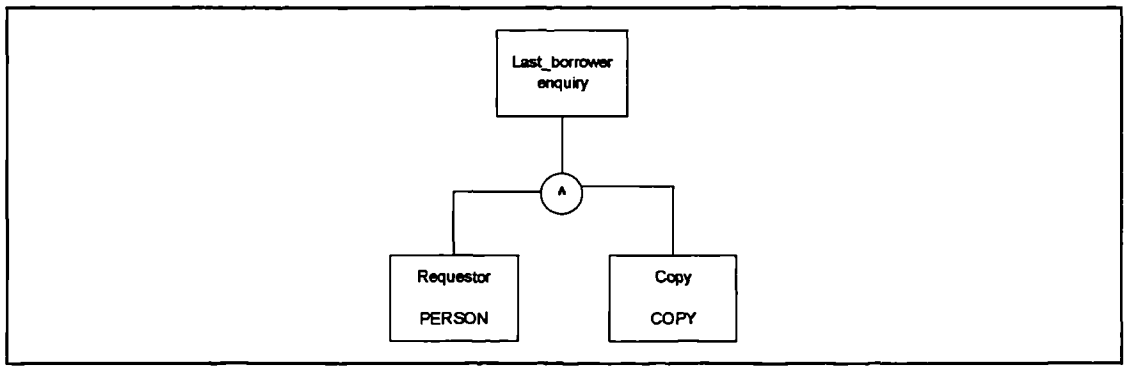


Figure 3-10: The RDSD of the data flow "Last_borrower_enquiry"

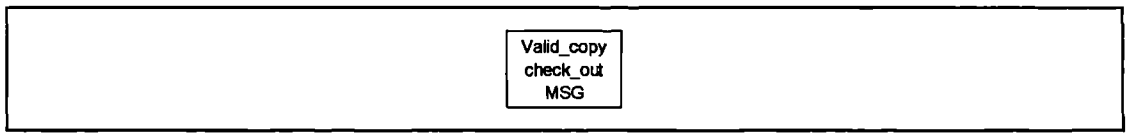


Figure 3-11: The RDSD of the data flow "Valid_copy_check_out"

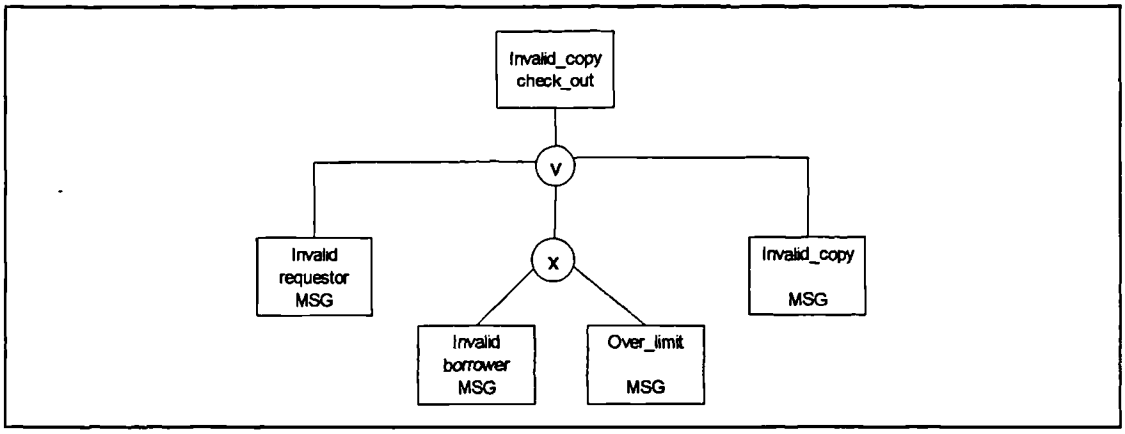


Figure 3-12: The RDSD of the data flow "Invalid_copy_check_out"

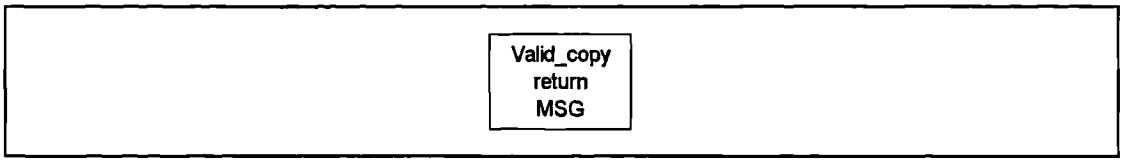


Figure 3-13: The RDSD of the data flow "Valid_copy_return"

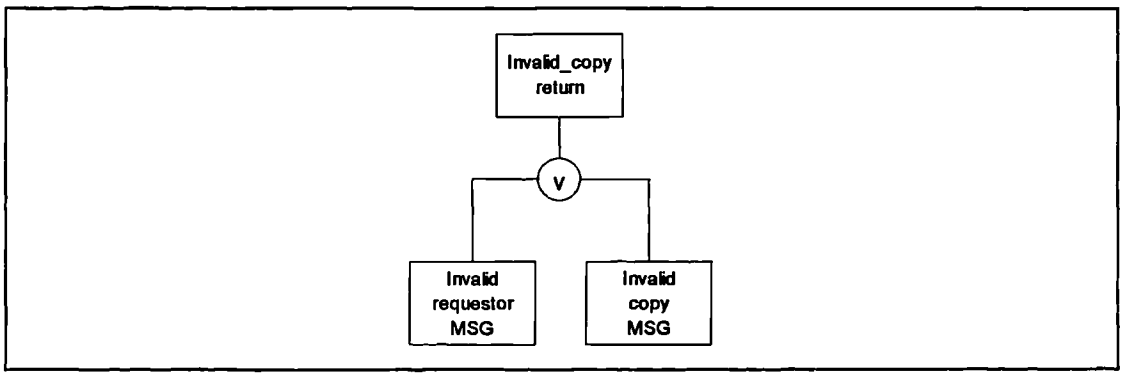


Figure 3-14: The RDSD of the data flow "Invalid_copy_return"

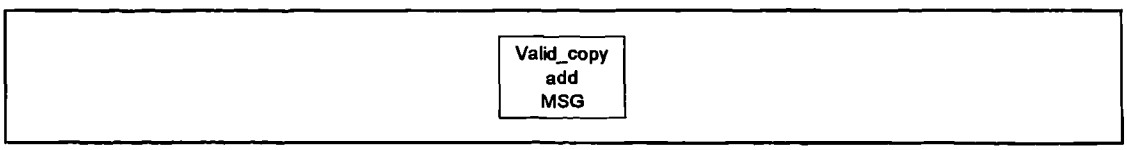


Figure 3-15: The RDSD of the data flow "Valid_copy_add"

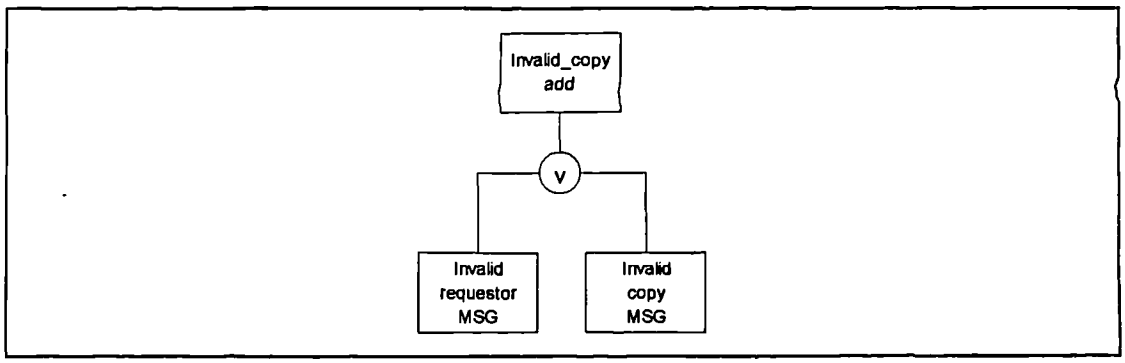


Figure 3-16: The RDSD of the data flow "Invalid_copy_add"

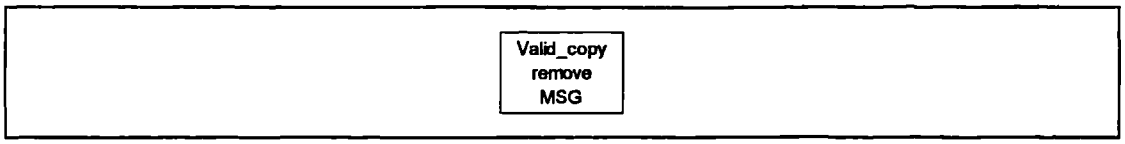


Figure 3-17: The RDSD of the data flow "Valid_copy_remove"

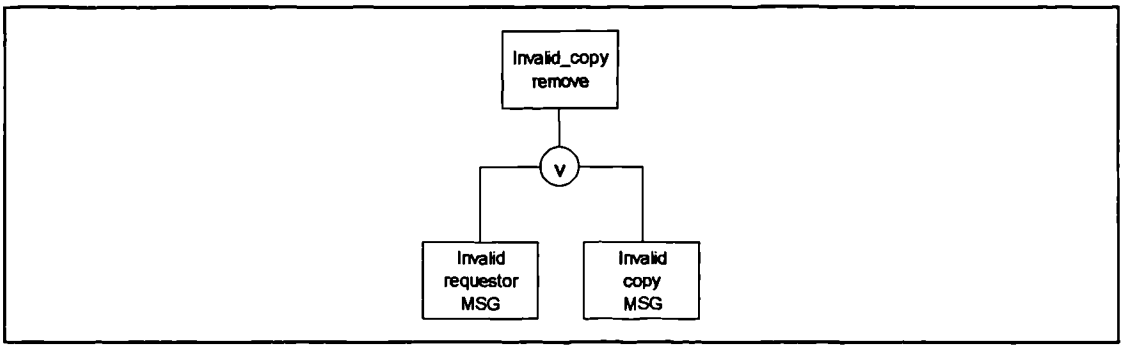


Figure 3-18: The RDSD of the data flow "Invalid_copy_remove"

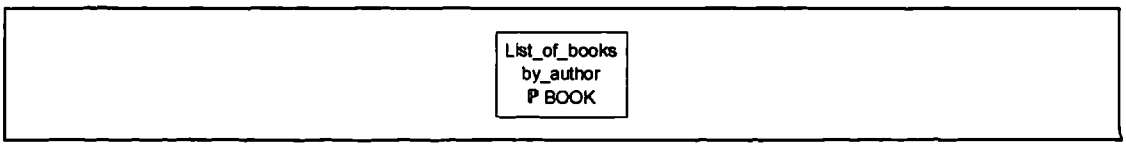


Figure 3-19: The RDSD of the data flow "List_of_books_by_author"

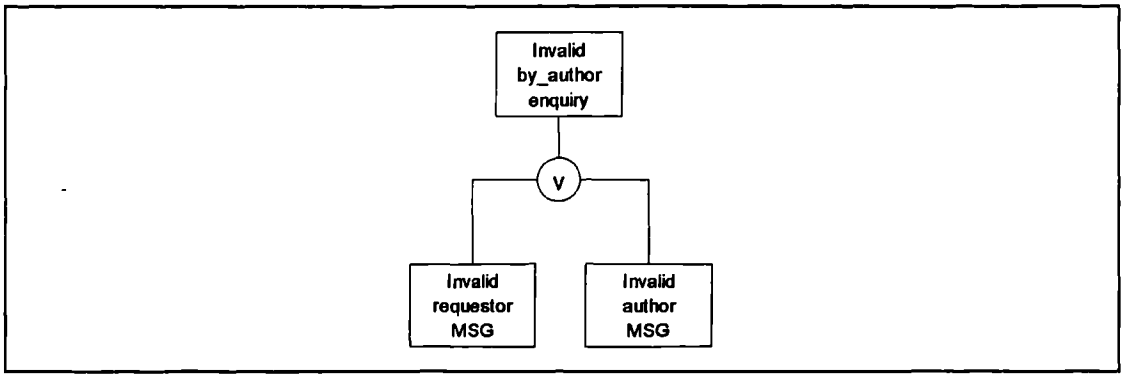


Figure 3-20: The RDSD of the data flow "Invalid_by_author_enquiry"

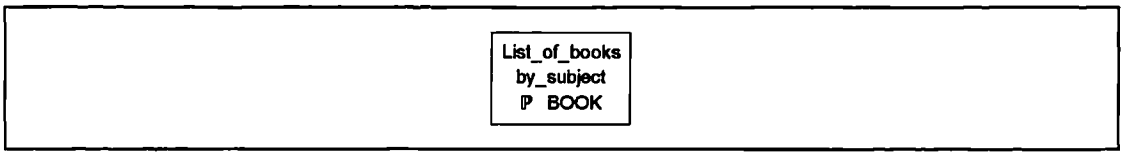


Figure 3-21: The RDSD of the data flow "List_of_books_by_subject"

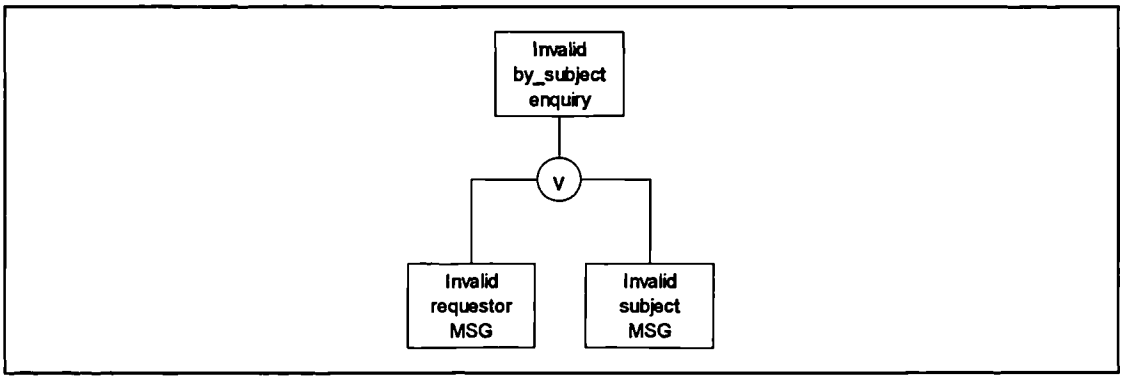


Figure 3-22: The RDSD of the data flow "Invalid_by_subject_enquiry"

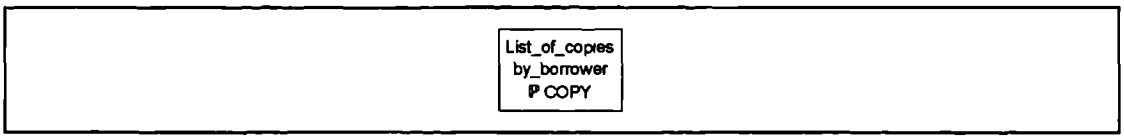


Figure 3-23: The RDSD of the data flow "List_of_copies_by_borrower"

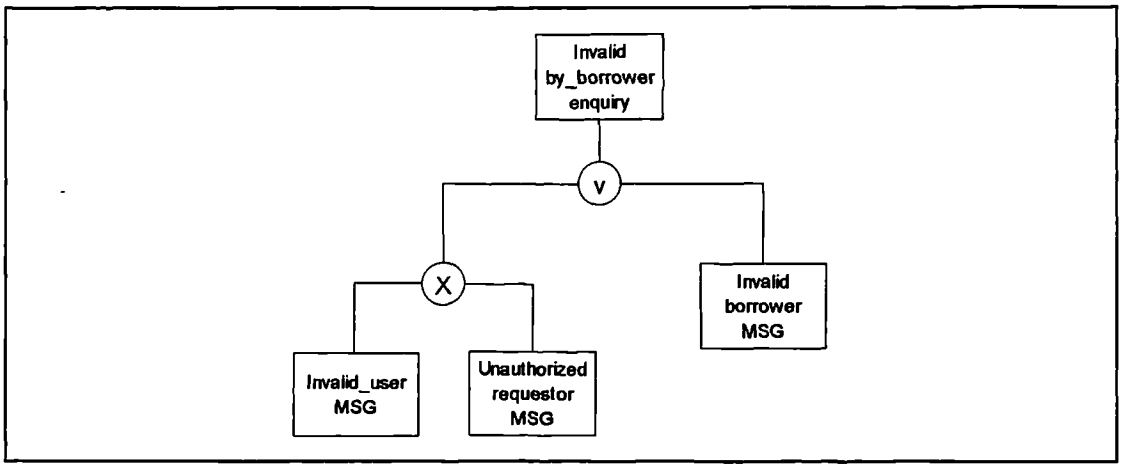


Figure 3-24: The RDSD of the data flow "Invalid_by_borrower_enquiry"

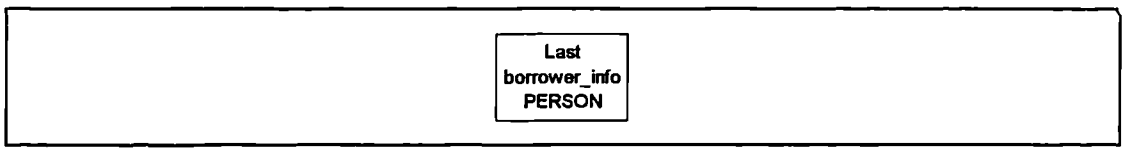


Figure 3-25: The RDSD of the data flow "Last_borrower_info"

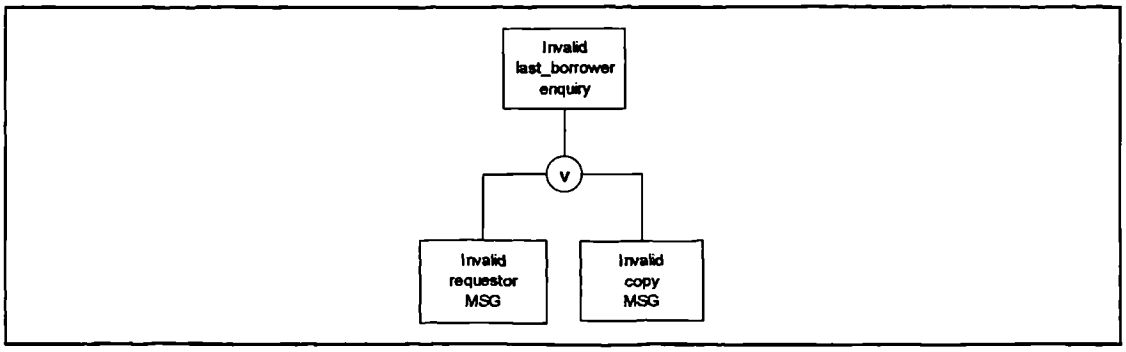


Figure 3-26: The RDSD of the data flow "Invalid_last_borrower_enquiry"

3.4.2 Step 2.2: draw the next level RDFDs and RDSDs

Steps 2.2.1 to 2.2.5 are repeated for each next level RDFD and RDSDs.

3.4.2.1 Step 2.2.1: identify sub-processes

The process on an RDFD can be decomposed into two or more sub-processes. If the process is decomposed, sub-processes must be identified.

The process 0 "Library" is decomposed into five sub-processes: "Check_out_copy", "Return_copy", "Add_copy", "Remove_copy", and "Enquiry".

3.4.2.2 Step 2.2.2: identify input and output data flows of each sub-process

First, the input and output data flows of the parent process must be assigned to each sub-process. In addition, the sub-processes might produce new internal data flows which are passed to other sub-processes on the same RDFD.

For example, these input and output data flows are assigned to the sub-process "Check_out_copy": "Copy_check_out", "Valid_copy_check_out", and "Invalid_copy_check_out".

There is no new data flow produced by five sub-processes mentioned above.

3.4.2.3 Step 2.2.3: draw the next level RDFD

Then, draw the next level RDFD.

Fig 3-27 shows the decomposition of the process 0 "Library" into five sub-processes.

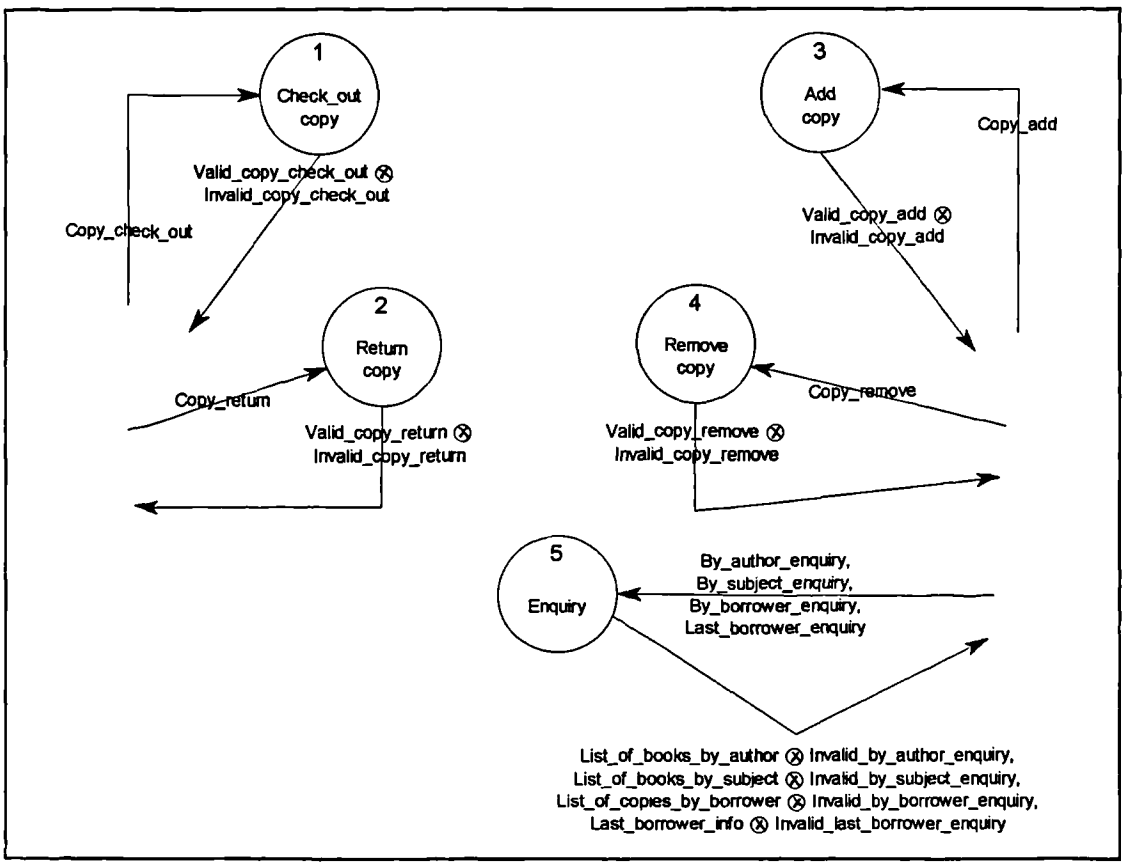


Figure 3-27: The decomposition of the process 0 RDFD

3.4.2.4 Step 2.2.4: identify the data components of each new internal data flow

When a process is decomposed into sub-processes, the sub-processes might produce internal data flows which are passed to other sub-processes on the same RDFD. These new internal data flows must be identified and data types are assigned.

The RDFD shown in Figure 3-27 has no internal data flow. However, in Figure 3-28, the process 1_1 "Check_copy_check_out" produces one internal data flow "Valid_copy_check_out_check". Then a data type is assigned to this internal data flow.

3.4.2.5 Step 2.2.5: draw RDSDs

Then, draw RDSDs of the new internal data flows identified in step 2.2.4.

Steps 2.2.1 to 2.2.5 are repeated until the details desired are obtained.

The remaining RDFDs and RDSDs of the library system are shown as follows.

A. Check out a copy of a book

Figure 3-28 shows the decomposition of the process 1 "Check_out_copy" into two sub-processes: the process 1.1 "Check_copy_check_out" and the process 1.2 "Record_copy_check_out". The process 1.1 receives the input data flow "Copy_check_out" from the process 1; it then produces either the output data flow "Invalid_copy_check_out" or "Valid_copy_check_out_check" but not both. The output data flow "Invalid_copy_check_out" is sent back to the process 1 whereas the output data flow "Valid_copy_check_out_check" is sent to the process 1.2.

The process 1.2 receives the input data flow "Valid_copy_check_out_check" from the process 1.1 and also receives two input data flows "Borrower" and "Copy" from the process 1. These three input data flows are all required (they are in conjunction). The process 1.2 then produces the output data flow "Valid_copy_check_out" which is sent back to the process 1. The process 1.2 deletes the checked out copy from the data store "Available_copies", creates (adds) the checked out copy into the data store "Checked_out_copies", and creates (adds) a relation into the data store "Currently_checks_out".

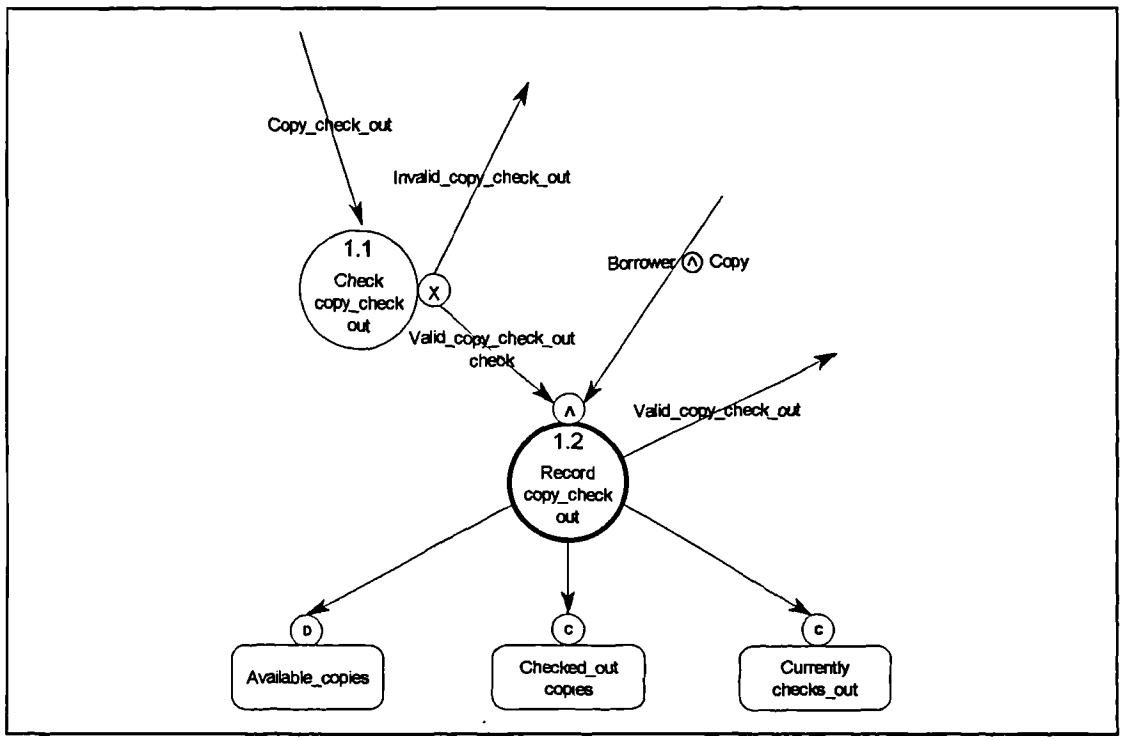


Figure 3-28: The decomposition of the process 1 RDFD

Since the data flow "Valid_copy_check_out_check" (see Figure 2-28) is a new internal data flow, an RDSD of this data flow is drawn as shown in Figure 3-29.

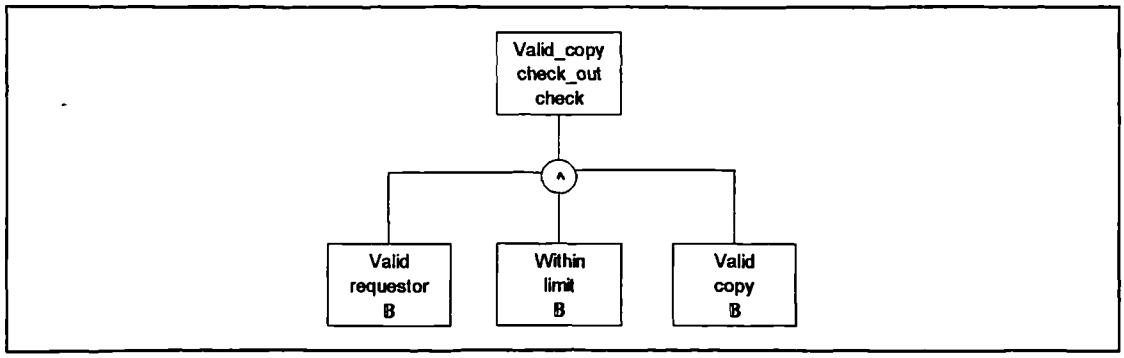


Figure 3-29: The RDSD of the data flow "Valid_check_copy_check_out"

Figure 3-30 shows the decomposition of the process 1.1. The process 1.1.1 checks whether the requestor is a valid requestor (the requestor is a member of staff). The process 1.1.2 checks whether the borrower is a valid borrower. If the borrower is a valid borrower, we need to further check that the borrower does not borrow more than the maximum number of copies allowed. The process 1.1.4 checks whether the copy is a valid copy (the copy is available to be borrowed).

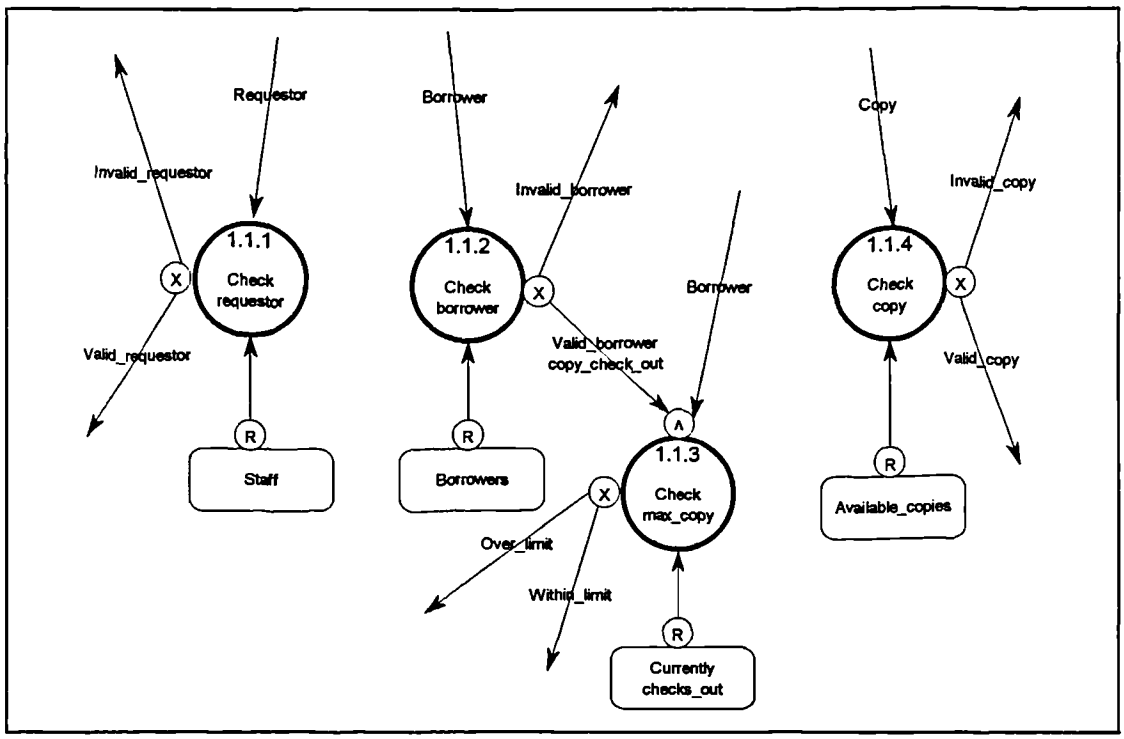


Figure 3-30: The decomposition of process 1.1 RDFD

Figure 3-31 shows the RDSD of the new internal data flow "Valid_borrower_copy_check_out" produced by the process 1.1.2.

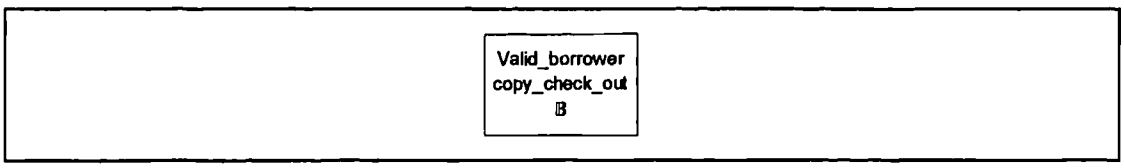


Figure 3-31: The RDSD of the data flow "Valid_borrower_copy_check_out"

B. Return a copy of a book

The decomposition of the process 2 is shown in Figure 3-32. The data structures of two new internal data flows produced by the processes 2.1 and 2.2 (see Figure 3-32) are shown in Figures 3-33 and 3-34.

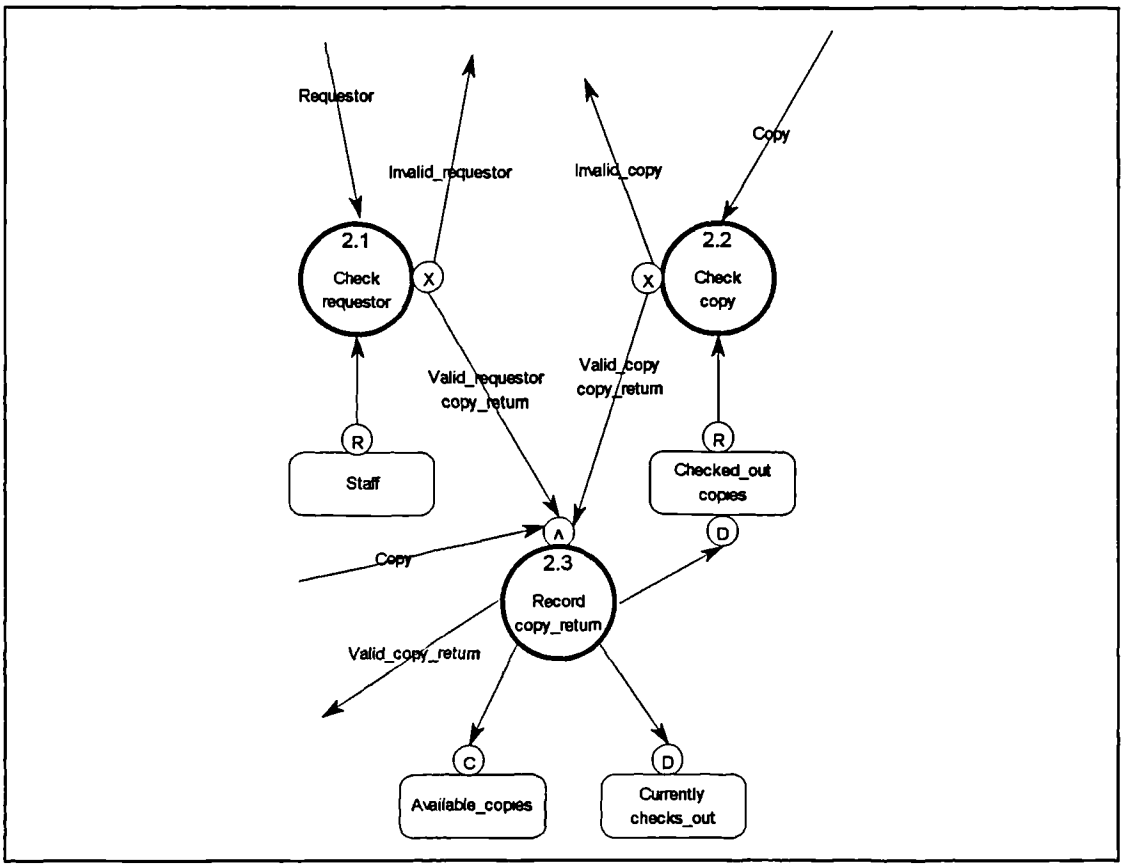


Figure 3-32: The decomposition of the process 2 RDFD

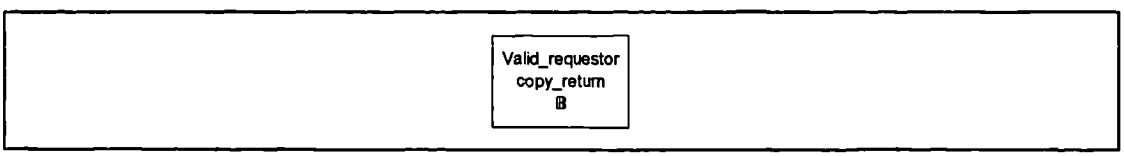


Figure 3-33: The RDSD of the data flow "Valid_requestor_copy_return"

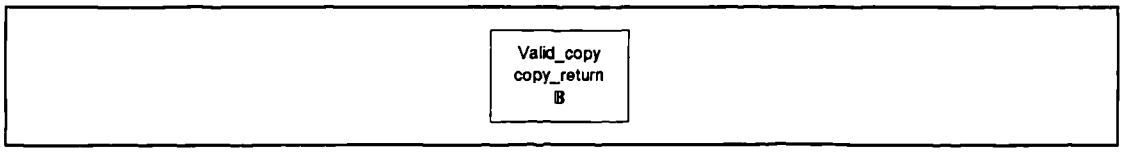


Figure 3-34: The RDSD of the data flow "Valid_copy_copy_return"

C. Add a copy of a book to the library

The decomposition of the process 3 is shown in Figure 3-35. The data structures of five new internal data flows (see Figure 3-35) are shown in Figures 3-36 to 3-40.

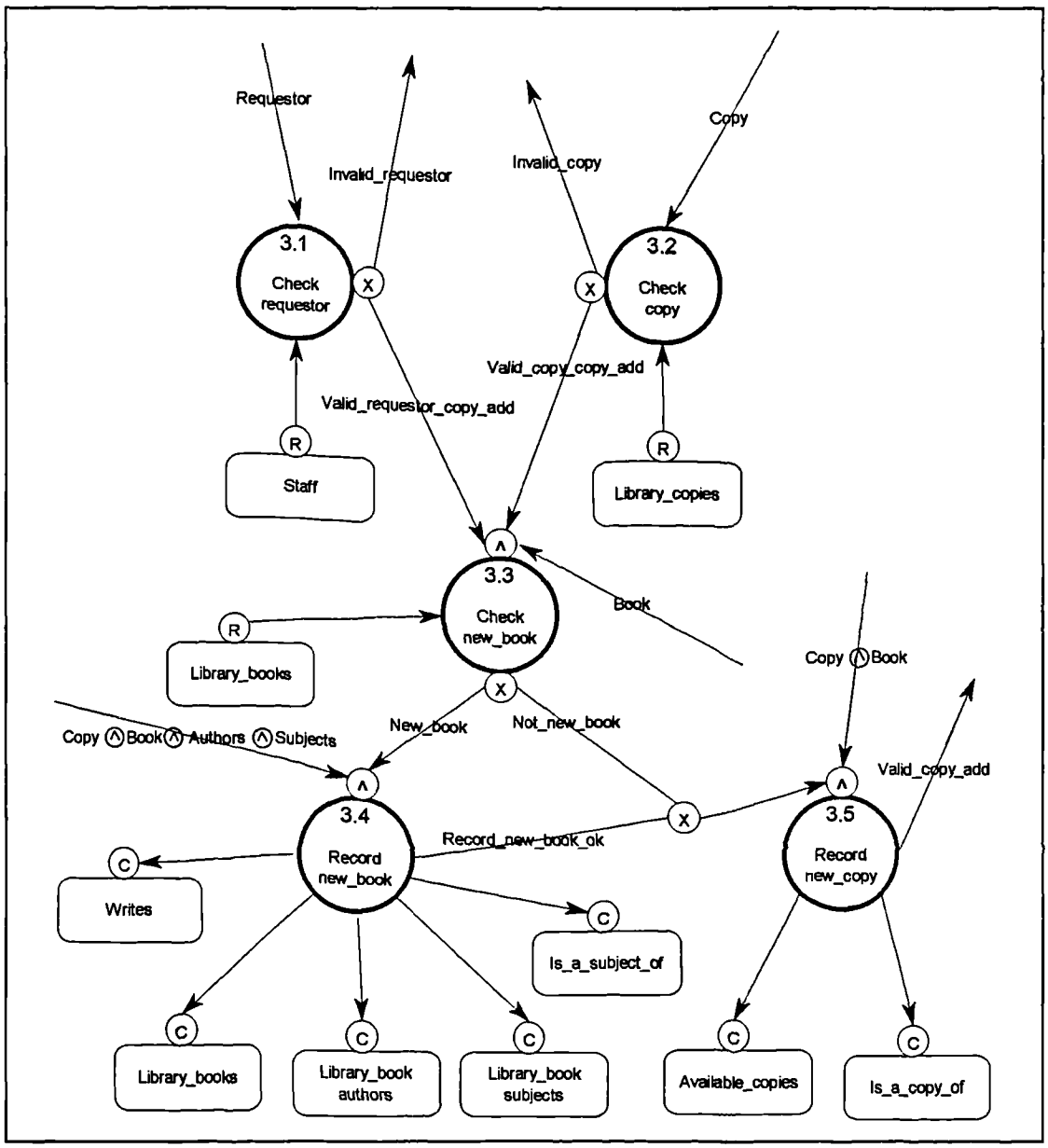


Figure 3-35: The decomposition of the process 3 RDFD

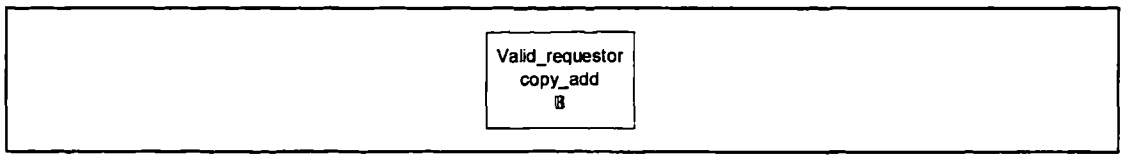


Figure 3-36: The RDSD of the data flow "Valid_requestor_copy_add"

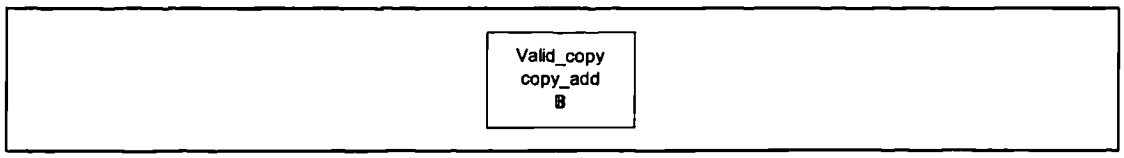


Figure 3-37: The RDSD of the data flow "Valid_copy_copy_add"

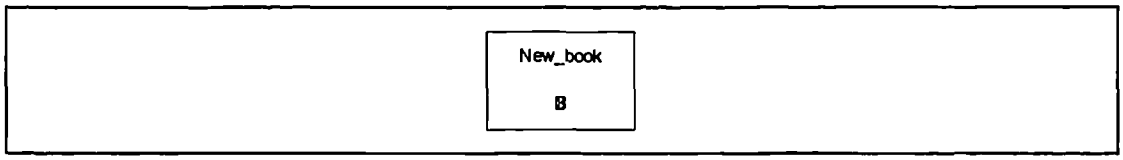


Figure 3-38: The RDSD of the data flow "New_book"

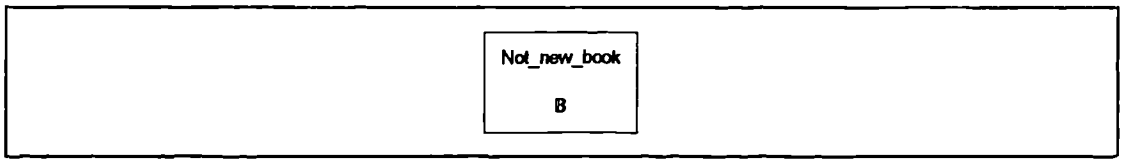


Figure 3-39: The RDSD of the data flow "Not_new_book"

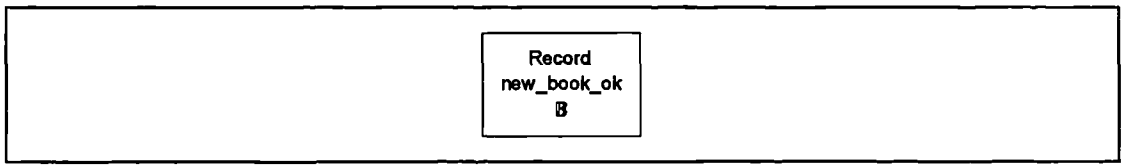


Figure 3-40: The RDSD of the data flow "Record_new_book_ok"

D. Remove a copy of a book from the library

The decomposition of the process 4 is shown in Figure 3-41. The data structures of five new internal data flows (see Figure 3-41) are shown in Figures 3-42 to 3-46.

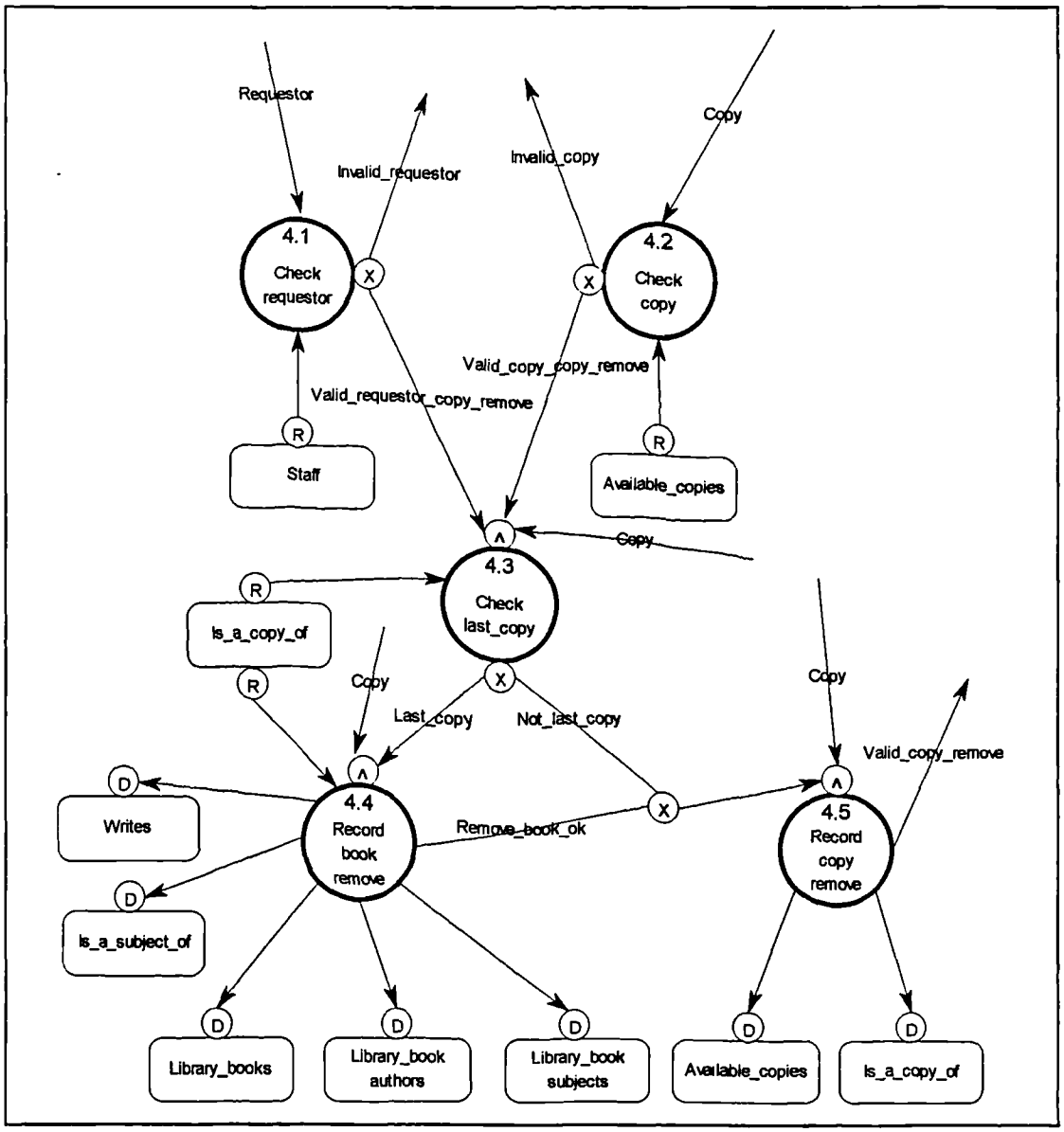


Figure 3-41: The decomposition of the process 4 RDFD

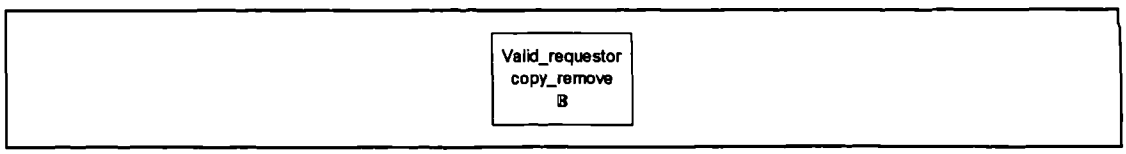


Figure 3-42: The RDSD of the data flow "Valid_requestor_copy_remove"

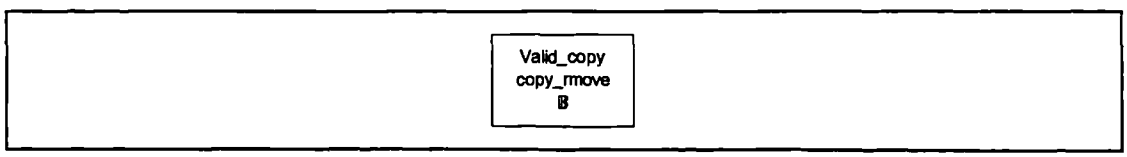


Figure 3-43: The RDSD of the data flow "Valid_copy_copy_remove"

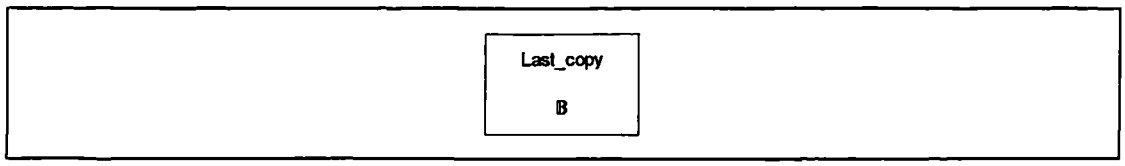


Figure 3-44: The RDSD of the data flow "Last_copy"

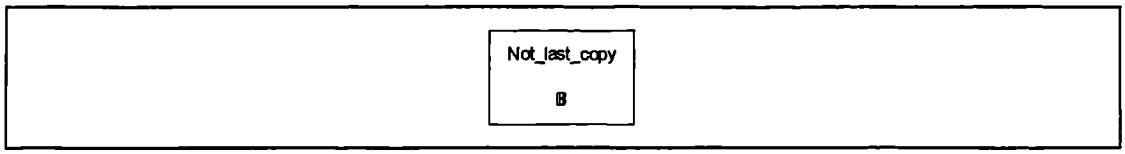


Figure 3-45: The RDSD of the data flow "Not_last_copy"

Figure 3-46: The RDSD of the data flow "Remove_book_ok"

E. Enquiry

The decomposition of the process 5 is shown in Figure 3-47.

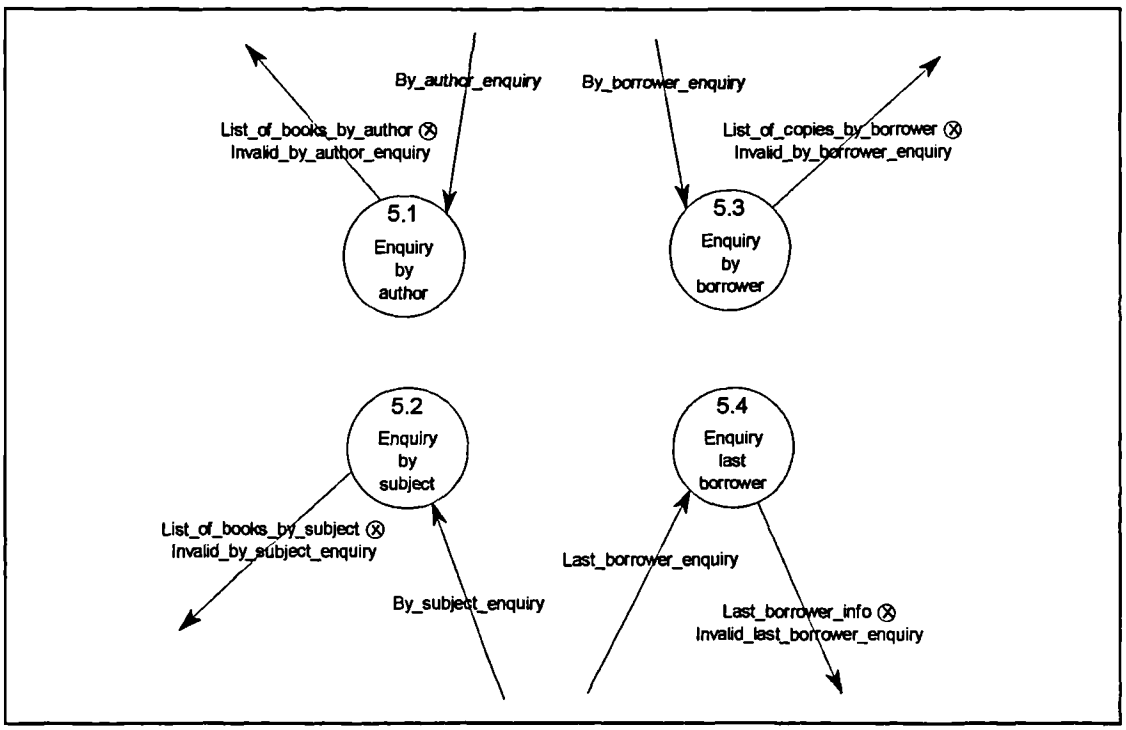


Figure 3-47: The decomposition of the process 5 RDFD

E.1 Enquire the list of books by a particular borrower

The decomposition of the process 5.1 is shown in Figure 3-48. The data structures of two new internal data flows (see Figure 3-48) are shown in Figures 3-49 and 3-50.

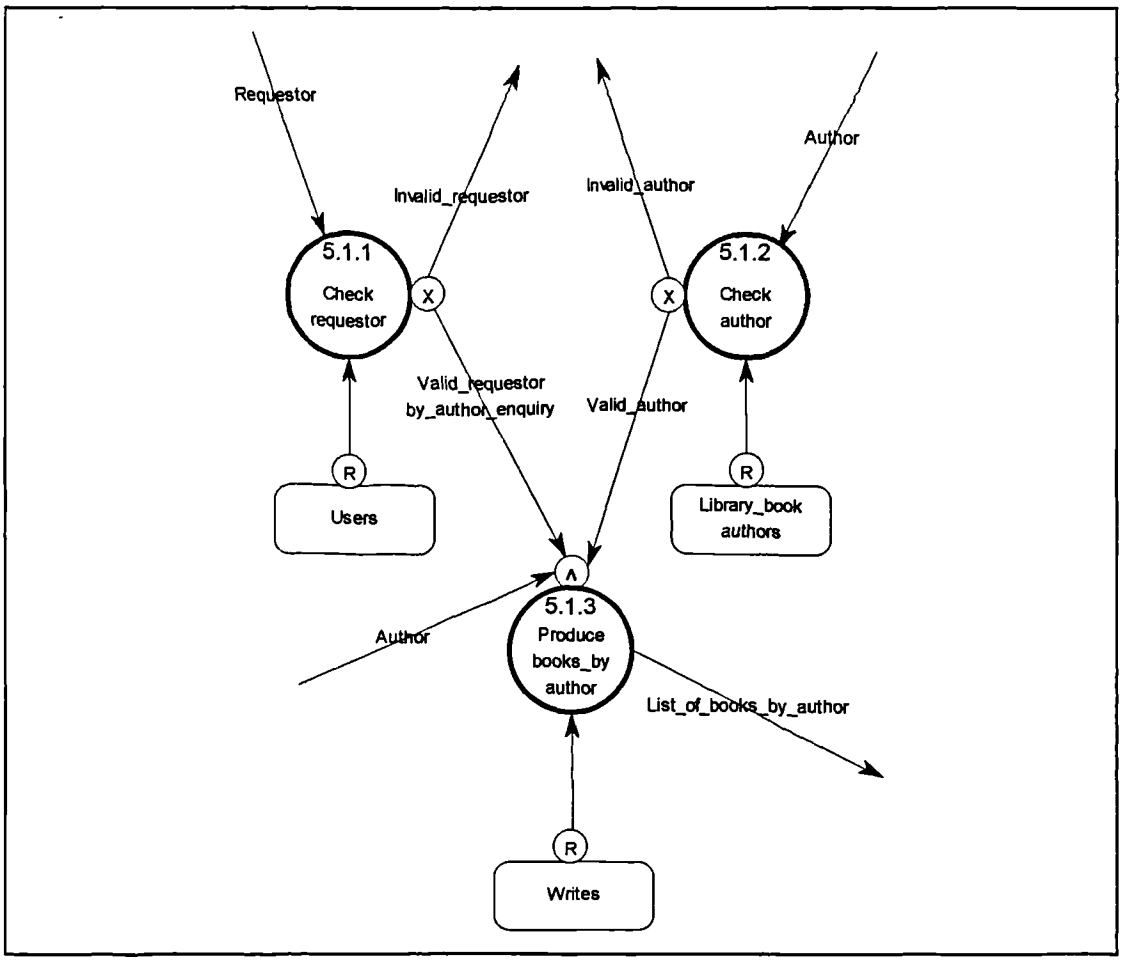


Figure 3-48: The decomposition of the process 5.1 RDFD

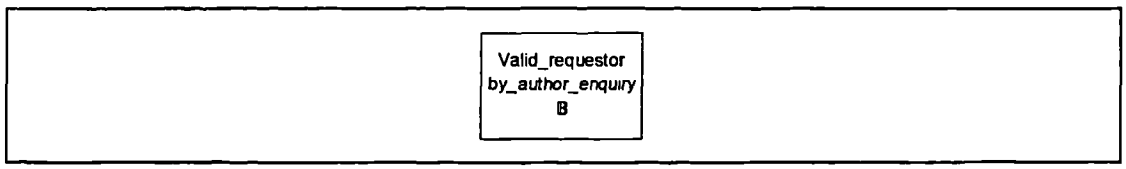


Figure 3-49: The RDSD of the data flow "Valid_requestor_by_author_enquiry"

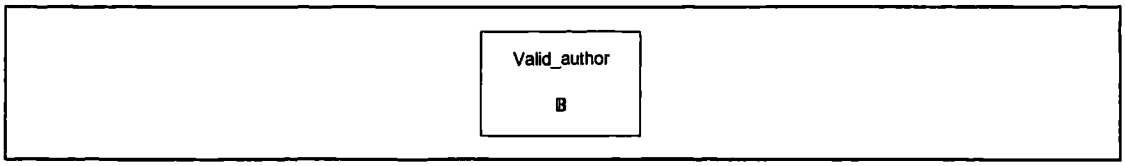


Figure 3-50: The RDSD of the data flow "Valid_author"

E.2 Enquire the list of books by a particular subject

The decomposition of the process 5.2 is shown in Figure 3-51. The data structures of two new internal data flows (see Figure 3-51) are shown in Figures 3-52 and 3-53.

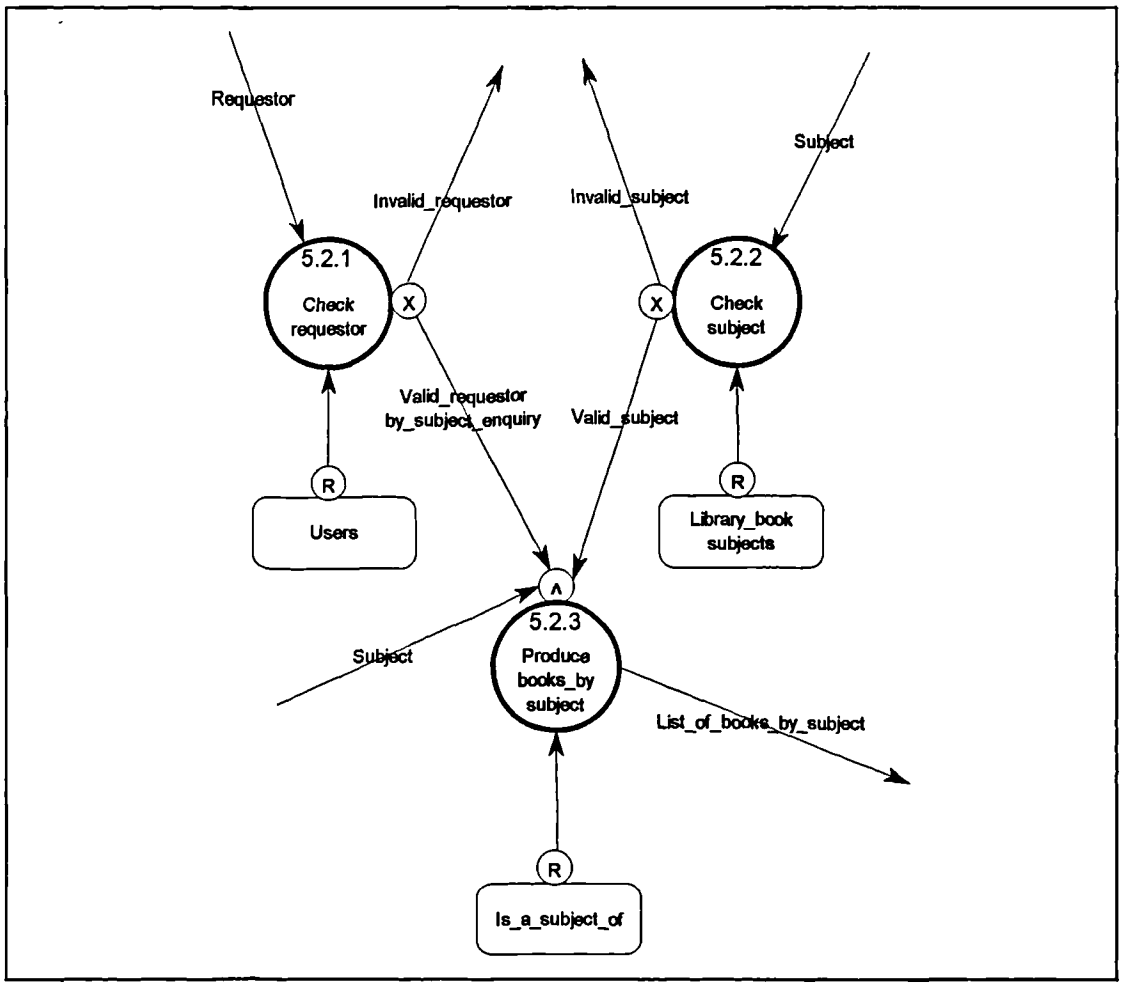


Figure 3-51: The decomposition of the process 5.2 RDFD

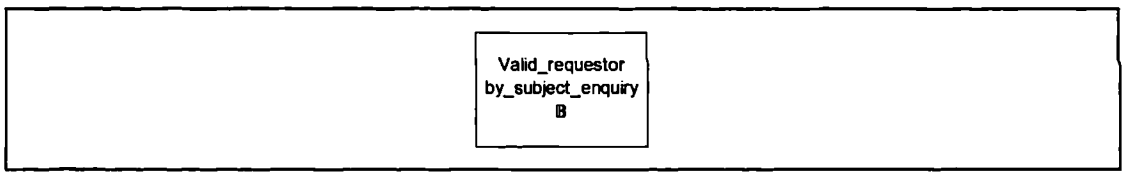


Figure 3-52: The RDSD of the data flow "Valid_requestor_by_subject_enquiry"

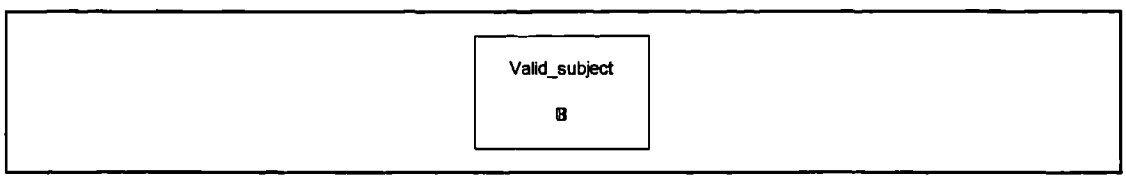


Figure 3-53: The RDSD of the data flow "Valid_subject"

E.3 Enquire the list of copies checked out by a particular borrower

The decomposition of the process 5.3 is shown in Figure 3-54. The data structures of five new internal data flows (see Figure 3-54) are shown in Figures 3-55 to 3-59.

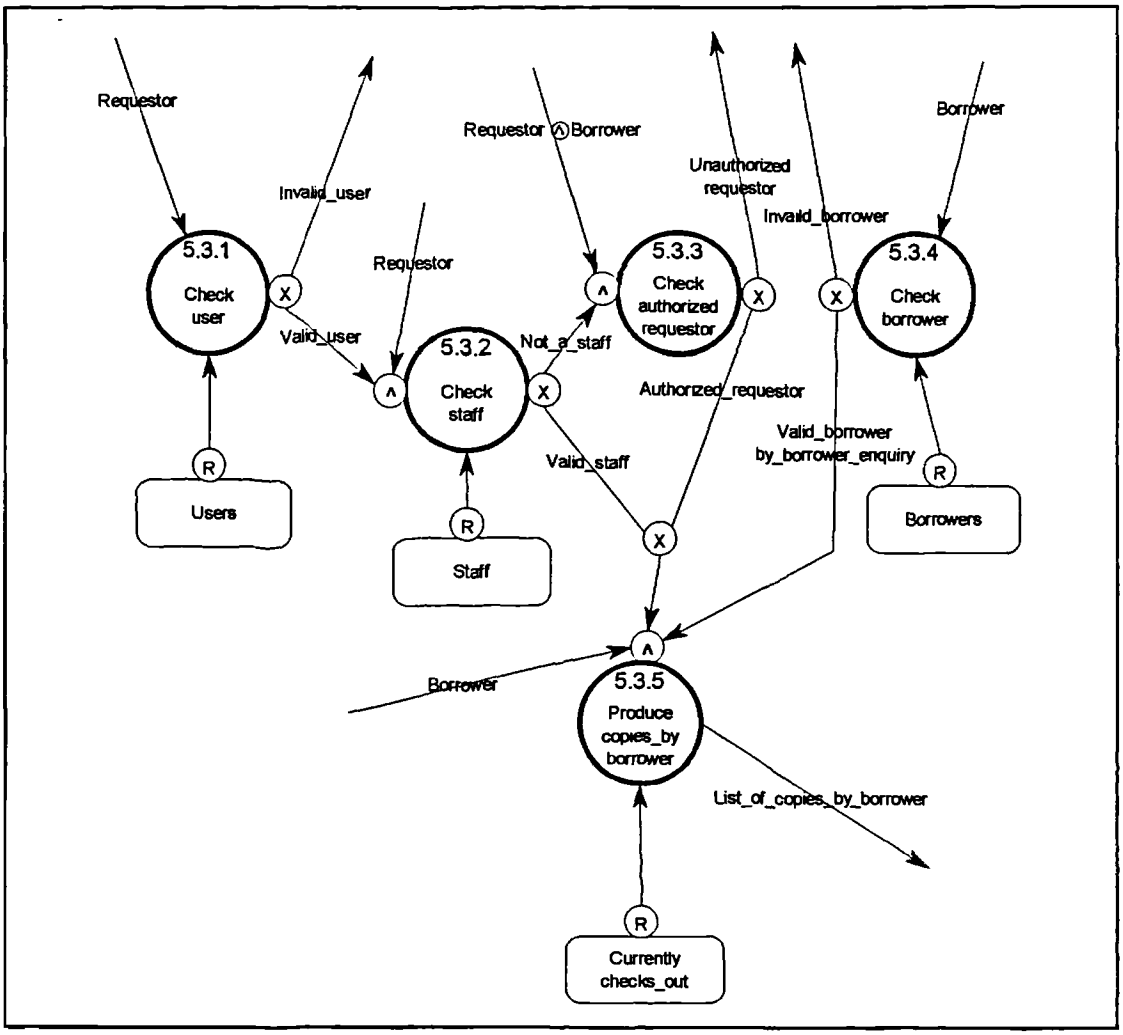


Figure 3-54: The decomposition of the process 5.3 RDFD

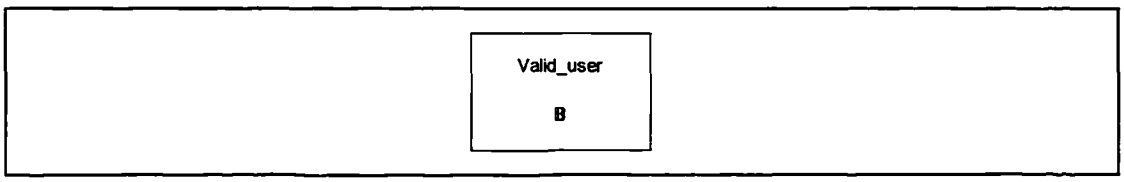


Figure 3-55: The RDSD of the data flow "Valid_user"

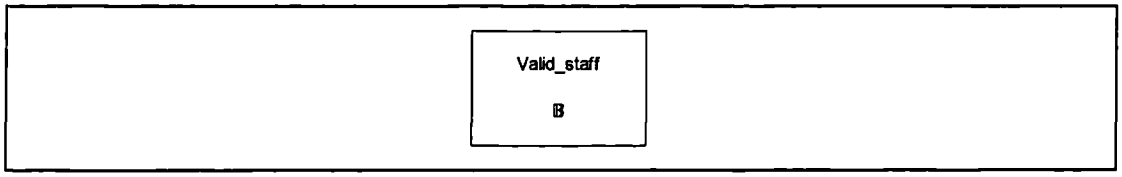


Figure 3-56: The RDSD of the data flow "Valid_staff"

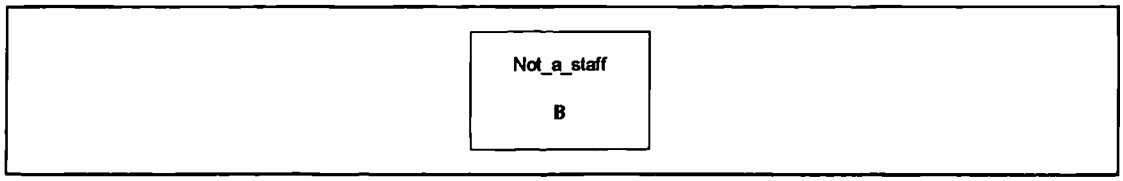


Figure 3-57: The RDSD of the data flow "Not_a_user"

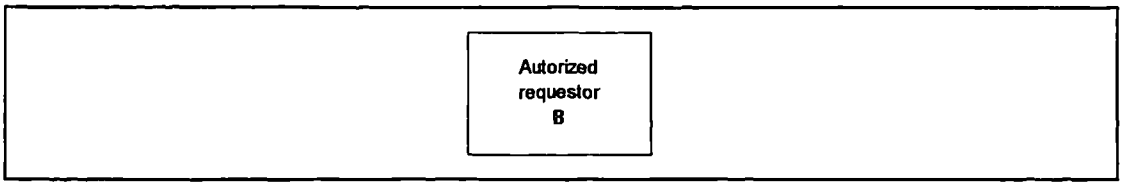


Figure 3-58: The RDSD of the data flow "Authorized_requestor"

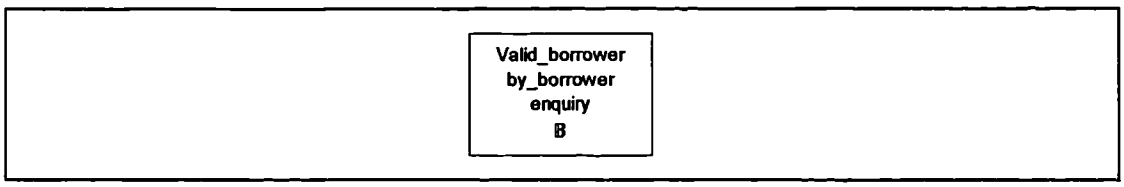


Figure 3-59: The RDSD of the data flow "Valid_borrower_by_borrower_enquiry"

E.4 Enquire what borrower last checked out a particular copy of a book

The decomposition of the process 5.4 is shown in Figure 3-60. The data structures of two new internal data flows (see Figure 3-60) are shown in Figures 3-61 and 3-62.

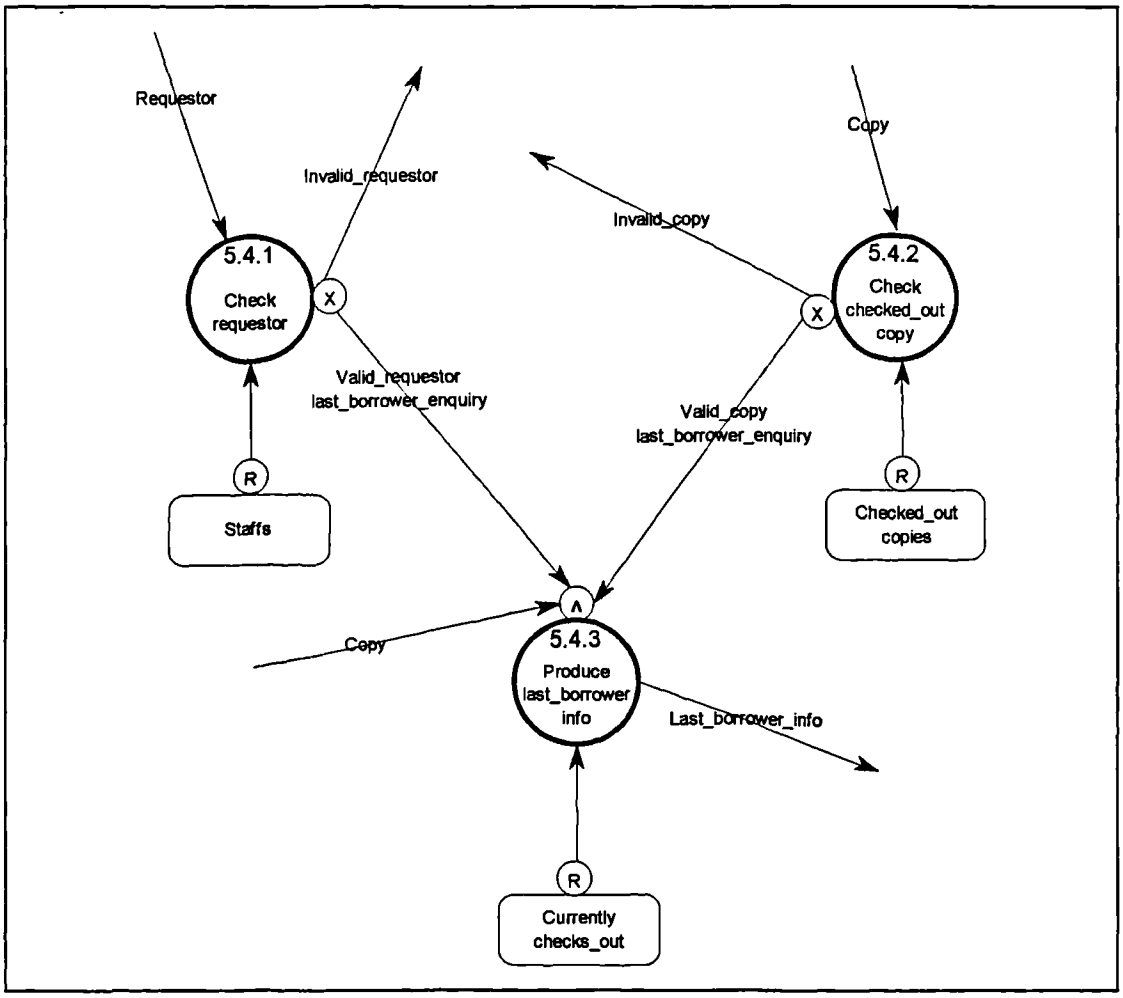


Figure 3-60: The decomposition of the process 5.4 RDFD

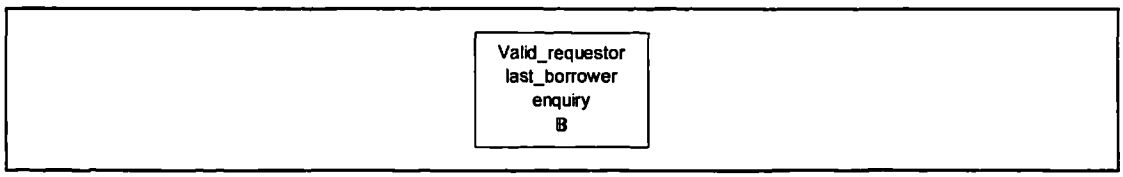


Figure 3-61: The RDSD of the data flow "Valid_requestor_last_borrower_enquiry"

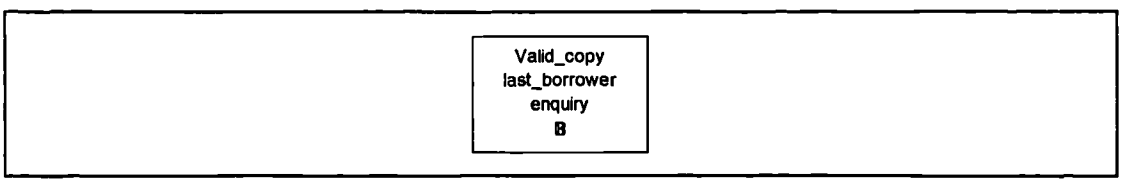


Figure 3-62: The RDSD of the data flow "Valid_copy_last_borrower_enquiry"

3.5 Step 3: write RZs

In this step, the software requirements are formally specified by writing Z specifications to capture both the static and dynamic aspects of a system as required by the end-users. The specifications are written in the standard Z (as defined in [86]) and the extended Z subset (as described in section 2.5.1).

The Z specifications are developed from the RERDs, RDFDs, and RDSDs produced earlier (as described in sections 3.2, 3.3, and 3.4). However, there are some requirements which cannot be captured by those three diagrams but must be specified in the Z specifications, therefore the informal requirements specification needs to be consulted also.

The Z specifications are developed by using the following steps.

3.5.1 Step 3.1: define the state of the system

As described in section 3.2.1 the static aspects of a system include the state the system can occupy and the critical requirements that must hold in every state. Section 3.3 discusses how to draw RERDs and RDSDs to depict the static aspects of a system. This section will describe how to write Z specifications to define the static aspects of a system, or in other words to define the state of the system. The Z specifications are developed from the RERDs and RDSDs produced earlier.

A. Define all data types

First, data types of all entities must be specified in Z specifications. These data types are the entity types shown in the RERDs. Some of these entity types may be further defined on the RDSDs. If the entity type is further defined on the RDSD, it is translated into a schema type; otherwise it is just simply translated into a basic type.

From the RERD of the library system (see Figure 3-1), there are five data types shown on the diagram. The five data types are "PERSON", "AUTHOR", "SUBJECT", "BOOK", and "COPY". Since these data types are not further defined on the RDSDs, they are translated into basic types in Z as follows.

[PERSON, AUTHOR, SUBJECT, BOOK, COPY]

B. For entities which have the same type, write a schema to define the state of those entities.

From the RERDs, we can easily identify the entities which have the same type. Then, we write a schema to define the state of those entities.

From the RERD of the library system (see Figure 3-1), we then write five schemas as follows.

Person_state

$$Users : \mathbb{P}\ PERSON$$
$$Borrowers : \mathbb{P}\ PERSON$$
$$Staff : \mathbb{P}\ PERSON$$

$$\langle Borrowers, Staff \rangle \text{ partition } Users$$

Author_state

Library_book_authors : ℙ AUTHOR

Subject_state

Library_book_subjects : ℙ SUBJECT

Book_state

Library_books : ℙ BOOK

Copy_state

Library_copies : ℙ COPY
Checked_out_copies : ℙ COPY
Available_copies : ℙ COPY

⟨Checked_out_copies, Available_copies⟩ partition Library_copies

C. Write a schema to define the state of the system

Then we write a schema to define the total state of the system by following these steps.

1) Include all schema names developed in the previous step.
2) Define the relationships of the entities by using the relationship symbols (as described in section 2.5.1.1). The relationships are shown on the RERDs produced in step 1.2.
3) Define other state variables.
4) Define the remaining critical requirements (these critical requirements can be extracted from the informal requirements specification; they are not shown in the RERDs).

The schema which defines the state of the library system can be written as follows.

Library_state

Person_state
Author_state
Subject_state
Book_state
Copy_state
Writes : Library_book_authors •<<—>>• Library_books
Is_a_subject_of : Library_book_subjects •<<—>>• Library_books
Is_a_copy_of : Library_copies •<<—>• Library_books
Currently_checks_out : Borrowers o<—>>• Checked_out_copies
Max_copy : $\mathbb{N}_1$

∀b : dom Currently_checks_out • # Currently_checks_out (| {b} |) ≤ Max_copy

The five schemas produced in the previous step are included. Then, the relationships of the entities as shown on the RERD (Figure 3-1) are defined. Next, the state variable "Max_copy" is defined. The state variable "Max_copy" is not shown on the RERD; it is stated in the informal requirements specification. Finally, in the predicate part, we define the remaining critical requirement, "a borrower may not have more than a predefined number of copies checked out at one time". Again this step uses information that is only stated in the informal requirements specification.

3.5.2 Step 3.2: define the initial state of the system

The initial state of the system, or the state the system is in when it is first started, can be defined as follows.

Init_library_state

Library_state

Users = ∅
Borrowers = ∅
Staff = ∅
Library_book_authors = ∅
Library_book_subjects = ∅
Library_books = ∅
Library_copies = ∅
Checked_out_copies = ∅
Available_copies = ∅
Writes = ∅
Is_a_subject_of = ∅
Is_a_copy_of = ∅
Currently_checks_out = ∅
Max_copy = 10

Assuming that a borrower is not allowed to borrow more than 10 copies at one time, therefore "Max_copy" is defined to 10.

3.5.3 Step 3.3: define the operations of the system

The operations of the system or in other words the dynamic aspects of the system are specified graphically by using RDFDs and RDSDs as already discussed in section 3.4. The operations of the system are also specified formally in Z specifications.

The Z specifications for defining the operations of the system are developed from the RDFDs and RDSDs produced in step 3.4. The RDFDs and RDSDs help ease the task of writing Z specifications since they provide a basis for structuring the Z specifications as well as providing information which can be translated easily into Z specifications. However, they cannot depict the detailed operations, therefore the informal requirements specification needs to be consulted as well.

For each process on the RDFDs, we write a schema to define that process. First, we write a schema to define the only process (the process 0) on the context diagram. However, before the process 0 is defined, some data types must be defined first. Then we write a schema to define each process on the next level RDFDs.

Concerning the library system, before the schema for defining the process 0 is written, the data type "MSG" needs to be defined as follows.

MSG ::= Copy_check_out_ok
| Copy_check_out_invalid_requestor
| Copy_check_out_invalid_borrower
| Copy_check_out_over_limit
| Copy_check_out_invalid_copy
| Copy_return_ok
| Copy_return_invalid_requestor
| Copy_return_invalid_copy
| Copy_add_ok
| Copy_add_invalid_requestor
| Copy_add_invalid_copy
| Copy_remove_ok
| Copy_remove_invalid_requestor
| Copy_remove_invalid_copy
| By_author_enquiry_invalid_requestor
| By_author_enquiry_invalid_author
| By_subject_enquiry_invalid_requestor
| By_subject_enquiry_invalid_subject
| By_borrower_enquiry_invalid_user
| By_borrower_enquiry_unauthorized_requestor
| By_borrower_enquiry_invalid_borrower
| Last_borrower_enquiry_invalid_requestor
| Last_borrower_enquiry_invalid_copy

Concerning the RDSDs drawn in the previous steps, there is the data type "MSG" and this data type is not shown on the RERD. Therefore, it must be defined. It is defined as a free type as shown above.

Next, we write a schema to define the process 0 (Figure 3-2).

In the declaration part of this schema, first all input and output data flows are defined. The RDSDs (Figures 3-3 to 3-26) provide the details of the structure of these data flows, and these RDSDs can be translated easily into Z specifications. Then we define the external entities and the inputs and/or outputs from/to each external entity.

In the predicate part of this schema, we write a predicate to state the relationship of the input and output data flows.

The schema to define the process 0 of the library system can be written as follows. The RDFD shown in Figure 3-2 and the RDSDs shown in Figures 3-3 to 3-26 provide the details for writing this schema.

For example:

"Copy_check_out? : (Requestor : PERSON⍟Borrower : PERSON⍟Copy : COPY)"

comes directly from Figure 3-3 and so on.

R0_Library

Copy_check_out? : (Requestor : PERSON ⍟ Borrower : PERSON ⍟
Copy : COPY)
Copy_return? : (Requestor : PERSON ⍟ Copy : COPY)
Copy_add? : (Requestor : PERSON ⍟ Copy : COPY ⍟ Book : BOOK ⍟
[Authors : ℙ AUTHOR] ⍟ [Subjects : ℙ SUBJECT])

Copy_remove? : (Requestor : PERSON ⓐ Copy : COPY)
By_author_enquiry? : (Requestor : PERSON ⓐ Author : AUTHOR)
By_subject_enquiry? : (Requestor : PERSON ⓐ Subject : SUBJECT)
By_borrower_enquiry? : (Requestor : PERSON ⓐ Borrower : PERSON)
Last_borrower_enquiry? : (Requestor : PERSON ⓐ Copy : COPY)
Valid_copy_check_out! : MSG
Invalid_copy_check_out! : (Invalid_requestor : MSG ⓥ
(Invalid_borrower : MSG ⓧ Over_limit : MSG) ⓥ Invalid_copy : MSG)
Valid_copy_return! : MSG
Invalid_copy_return! : (Invalid_requestor : MSG ⓥ Invalid_copy : MSG)
Valid_copy_add! : MSG
Invalid_copy_add! : (Invalid_requestor : MSG ⓥ Invalid_copy : MSG)
Valid_copy_remove! : MSG
Invalid_copy_remove! : (Invalid_requestor : MSG ⓥ Invalid_copy : MSG)
List_of_books_by_author! : ℙ BOOK
Invalid_by_author_enquiry! : (Invalid_requestor : MSG ⓥ Invalid_author : MSG)
List_of_books_by_subject! : ℙ BOOK
Invalid_by_subject_enquiry! : (Invalid_requestor : MSG ⓥ Invalid_subject : MSG)
List_of_copies_by_borrower! : ℙ COPY
Invalid_by_borrower_enquiry! : ((Invalid_user : MSG ⓧ
Unauthorized_requestor : MSG) ⓥ Invalid_borrower : MSG)
Last_borrower_info! : PERSON
Invalid_last_borrower_enquiry! : (Invalid_requestor : MSG ⓥ
Invalid_copy : MSG)
⇐ Borrowers_ext (Copy_check_out?, Copy_return?,
By_author_enquiry?, By_subject_enquiry?, By_borrower_enquiry?,
Valid_copy_check_out!, Invalid_copy_check_out!,
Valid_copy_return!, Invalid_copy_return!,
List_of_books_by_author!, Invalid_by_author_enquiry!,
List_of_books_by_subject!, Invalid_by_subject_enquiry!,
List_of_copies_by_borrower!, Invalid_by_borrower_enquiry!)
⇐ Staff_ext (Copy_add?, Copy_remove?,
By_author_enquiry?, By_subject_enquiry?, By_borrower_enquiry?,
Last_borrower_enquiry?,
Valid_copy_add!, Invalid_copy_add!,
Valid_copy_remove!, Invalid_copy_remove!,
List_of_books_by_author!, Invalid_by_author_enquiry!,
List_of_books_by_subject!, Invalid_by_subject_enquiry!,
List_of_copies_by_borrower!, Invalid_by_borrower_enquiry!,
Last_borrower_info!, Invalid_last_borrower_enquiry!)

Copy_check_out?, Copy_return?, Copy_add?,Copy_remove?,
By_author_enquiry?, By_subject_enquiry?, By_borrower_enquiry?,
Last_borrower_enquiry?
➲
Valid_copy_check_out!ⓧInvalid_copy_check_out!,
Valid_copy_return! ⓧ Invalid_copy_return!,
Valid_copy_add! ⓧ Invalid_copy_add!,
Valid_copy_remove! ⓧ Invalid_copy_remove!,
List_of books_by_author! ⓧ Invalid_by_author_enquiry!,
List_of_books_by_subject! ⓧ Invalid_by_subject_enquiry!,
List_of_copies_by_borrower!ⓧ Invalid_by_borrower_enquiry!,
Last_borrower_info! ⓧ Invalid_last_borrower_enquiry!

Next, the processes 1 to 5 (Figure 3-27) can be translated into Z specifications as follows.

R1_Check_out_copy

⇧ R0_Library (Copy_check_out?, Valid_copy_check_out!,
Invalid_copy_check_out!)

Copy_check_out? ➲ Valid_copy_check_out! ⊗ Invalid_copy_check_out!

R2_Return_copy

⇧ R0_Library (Copy_return?, Valid_copy_return!, Invalid_copy_return!)

Copy_return? ➲ Valid_copy_return! ⊗ Invalid_copy_return!

R3_Add_copy

⇧ R0_Library (Copy_add?, Valid_copy_add!, Invalid_copy_add!)

Copy_add? ➲ Valid_copy_add! ⊗ Invalid_copy_add!

R4_Remove_copy

⇧ R0_Library (Copy_remove?, Valid_copy_remove!, Invalid_copy_remove!)

Copy_remove? ➲ Valid_copy_remove! ⊗ Invalid_copy_remove!

R5_Enquiry

⇧ R0_Library (By_author_enquiry?, By_subject_enquiry?,
By_borrower_enquiry?, Last_borrower_enquiry?,
List_of_books_by_author!, Invalid_by_author_enquiry!,
List_of_books_by_subject!, Invalid_by_subject_enquiry!,
List_of_copies_by_borrower!, Invalid_by_borrower_enquiry!,
Last_borrower_info!, Invalid_last_borrower_enquiry!)

By_author_enquiry?, By_subject_enquiry?, By_borrower_enquiry?,
Last_borrower_enquiry? ➲
List_of_books_by_author! ⊗ Invalid_by_author_enquiry!,
List_of_books_by_subject! ⊗ Invalid_by_subject_enquiry!,
List_of_copies_by_borrower! ⊗ Invalid_by_borrower_enquiry!,
Last_borrower_info! ⊗ Invalid_last_borrower_enquiry!

The schema "R1_Check_out_copy" can be described as follows. In its declaration part, we define where the inputs and outputs of the schema are from. As defined in this schema, the input "Copy_check_out?" is from the schema "R0_Library" and the outputs "Valid_copy_check_out!" and "Invalid_copy_check_out!" are passed to the schema "R0_library". In its predicate part, we define the relationship between the input and outputs. It defines that from the input "Copy_check_out?", the process will produce either "Valid_copy_check_out!" or "Invalid_copy_check_out!" but not both.

A. Check out a copy of a book

Next, the processes 1.1 and 1.2 (Figure 3-28), all sub-processes of the process 1.1 (Figure 3-30) and the related RDSDs (Figures 3-29 and 3-31) can be translated into Z specifications as follows.

R1_1_Check_copy_check_out

⇧ R1_Check_out_copy (Copy_check_out?, Invalid_copy_check_out!)
Valid_copy_check_out_check! : (Valid_requestor : 𝔹 Ⓐ Within_limit : 𝔹 Ⓐ
Valid_copy : 𝔹)

Copy_check_out? ➲ Valid_copy_check_out_check! Ⓧ Invalid__copy_check_out!

R1_2_Record_copy_check_out

⇧ R1_Check_out_copy (Borrower?, Copy?, Valid_copy_check_out!)
⇨ R1_1_Check_copy_check_out (Valid_copy_check_out_check?)
Library_state (Δ Available_copies, Δ Checked_out_copies, Δ Currently_checks_out)

Borrower? Ⓐ Copy? Ⓐ Valid_copy_check_out_check? ➲ Valid_copy_check_out!
Available_copies ' = Available_copies \ {Copy?}
Checked_out_copies ' = Checked_out_copies ∪ {Copy?}
Currently _checks_out ' = Currently_checks_out ∪ {Borrower? ↦ Copy?}
Valid_copy_check_out! = Copy_check_out_ok

R1_1_1_Check_requestor

⇧ R1_1_Check_copy_check_out (Requestor?, Valid_requestor!, Invalid_requestor!)
Library_state (Ξ Staff)

Requestor? ➲ Valid_requestor! Ⓧ Invalid_requestor!
(Requestor? ∈ Staff ∧ Valid_requestor! = True)
∨ (Requestor? ∉ Staff ∧ Invalid_requestor! = Copy_check_out_invalid_requestor)

R1_1_2_Check_borrower

⇧ R1_1_Check_copy_check_out (Borrower?, Invalid_borrower!)
Library_state (Ξ Borrowers)
Valid_borrower_copy_check_out! : 𝔹

Borrower? ➲ Valid_borrower_copy_check_out! Ⓧ Invalid__borrower!
(Borrower? ∈ Borrowers ∧ Valid_borrower_copy_check_out! = True)
∨ (Borrower? ∉ Borrowers ∧ Invalid_borrower! = Copy_check_out_invalid_borrower)

R1_1_3_Check_max_copy

⇑ R1_1_Check_copy_check_out (Borrower?, Within_limit!, Over_limit!)
⇨ R1_1_2_Check_borrower (Valid_borrower_copy_check_out?)
Library_state (Ξ Currently_checks_out, Ξ Max_copies)

Borrower? ⍟ Valid_borrower_copy_check_out?➲ Within_limit! ⊗ Over_limit!
((# Currently_checks_out (| {Borrower?} |) < Max_copy) ∧ Within_limit! = True)
∨ ((# Currently_checks_out (| {Borrower?} |) ≥ Max_copy) ∧
Over_limit! = Copy_check_out_over_limit)

R1_1_4_Check_copy

⇑ R1_1_Check_copy_check_out (Copy?, Valid_copy!, Invalid_copy!)
Library_state (Ξ Available_copies)

Copy? ➲ Valid_copy! ⊗ Invalid_copy!
(Copy? ∈ Available_copies ∧ Valid_copy! = True)
∨ (Copy? ∉ Available_copies ∧ Invalid_copy! = Copy_check_out_invalid_copy)

The schema "R1_1_Check_copy_check_out" receives one input "Copy_check_out?" from the schema "R1_Check_out_copy" and returns one output "Invalid_copy_check_out!" to the same schema. However, there are two outputs that can be produced by this schema as defined in the predicate. The other output is "Valid_copy_check_out_check!". This output is not defined by the parent schema (the schema "R1_Check_out_copy"), therefore it must be defined (by translating the RDSD of this output, shown in Figure 3-29, into a Z statement).

Since the schema "R1_2_Record_copy_check_out" defines the bottom level process, the detailed operations must be defined. This schema receives two inputs from and sends one output to the schema "R1_Check_out_copy". Also, this schema receives one input from and sends no output to the schema "R1_1_Check_copy_check_out".

This schema also states that it changes only "Available_copies", "Checked_out_copies", and "Currently_checks_out", but not other state variables of the "Library_state". The way the Δ and Ξ are used in this thesis is quite different from the general practice. We believe that it is more convenient and more suitable to use these two symbols this way.

In standard practice, the Δ notation as well as Ξ notation is followed by a schema name. For example, ΔState ("State" is the state space of a data type) is implicitly defined as the combination of State and State':

Δ State

State
State'

Like ΔState, the ΞState is implicitly define whenever a schema "State" is introduced as the state space of a data type:

ΞState

State
State'

θ State = θ State'

In this thesis, the Δ and Ξ conventions can be used in the standard way as mentioned above. However, they can also be used in a more specific way; either one of them can be used in front of a state variable (the state space of a data type may contain more than one state variable) to define specifically the effect to that state variable.

For example, from the schema "R1_2_Record_copy_check_out", the declaration statement

Library_state (Δ Available_copies, Δ Checked_out_copies, Δ Currently_checks_out)

defines that only these three state variables in the state space "Library_state" may be changed as the result of the operations defined in this schema but not other state variables.

For example, from the schema "R1_1_1_Check_requestor", the declaration statement

Library_state (Ξ Staff)

defines that this schema concerns only the state variable "Staff" in the state space "Library_state", it does not concern other state variables in the "Library_state". In addition, the before and after state of the state variable "Staff" are the same; in other words the operations of this schema do not change the state of the state variable "Staff".

In this thesis, we introduce the data type boolean "𝔹" as a built-in type. 𝔹 is the set of boolean values {True, False}.

In the schema "R1_1_Check_copy_check_out", the data flow "Valid_requestor" is assigned the data type "𝔹". In the schema "R1_1_1_Check_requestor", the data flow "Valid_requestor" is assigned to "True" if the requestor is a staff. In fact, the data flow "Valid_requestor" is a control flow generated by one process and then sent to another process to signal particular status or condition; the data value assigned to this control flow is not at all important. However, in this thesis, all control flows will be assigned the data type "𝔹".

In fact, the process of developing Z specifications to define the operations of the system can be done mechanically until the bottom-level processes are reached. The Z specifications for the higher-level processes can be automatically generated from the RDFDs and the related RDSDs. However, for the bottom-level processes only part of the Z specifications can be automatically generated; the specifiers are required to write detailed operations of the bottom-level processes themselves since the detailed operations are not captured by the RDFDs and RDSDs

B. Return a copy of a book

Next, the processes 2.1 to 2.3 (Figure 3-32) and the related RDSDs (Figures 3-33 to 3-34) can be translated into Z specifications as follows.

R2_1_Check_requestor

⇧ R2_Return_copy (Requestor?, Invalid_requestor!)
Library_state (Ξ Staff)
Valid_requestor_copy_return! : 𝔹

Requestor? ➲ Valid_requestor_copy_return! ⊗ Invalid_requestor!
(Requestor? ∈ Staff ∧ Valid_requestor_copy_return! = True)
∨ (Requestor? ∉ Staff ∧ Invalid_requestor! = Copy_return_invalid_requestor)

R2_2_Check_copy

⇧ R2_Return_copy (Copy?, Invalid_copy!)
Library_state (Ξ Checked_out_copies)
Valid_copy_copy_return! : 𝔹

Copy? ➲ Valid_copy_copy_return! ⊗ Invalid_copy!
(Copy? ∈ Checked_out_copies ∧ Valid_copy_copy_return! = True)
∨ (Copy? ∉ Checked_out_copies ∧ Invalid_copy! = Copy_return_invalid_copy)

R2_3_Record_copy_return

⇧ R2_Return_copy (Copy?, Valid_copy_return!)
⇨ R2_1_Check_requestor (Valid_requestor_copy_return?)
⇨ R2_2_Check_copy (Valid_copy_copy_return?)
Library_state (Δ Checked_out_copies, Δ Available_copies, Δ Currently_checks_out)

Copy?⍟ Valid_requestor_copy_return? ⍟ Valid_copy_copy_retrun? ➲
Valid_copy_return!
Checked_out_copies ' = Checked_out_copies \ {Copy?}
Available_copies ' = Available_copies ∪ {Copy?}
Currently_checks_out ' = Currently_checks_out ⩤{Copy?}
Valid_copy_return! = Copy_return_ok

C. Add a copy of a book to the library

The processes 3.1 to 3.5 (Figure 3-35) and the related RDSDs (Figures 3-36 to 3-40) can be translated into Z specifications as follows.

R3_1_Check_requestor

⇧ R3_Add_copy (Requestor?, Invalid_requestor!)
Library_state (Ξ Staff)
Valid_requestor_copy_add! : 𝔹

Requestor? ➲ Valid_requestor_copy_add! ⊗ Invalid_requestor!
(Requestor? ∈ Staff ∧ Valid_requestor_copy_add! = True)
∨ (Requestor? ∉ Staff ∧ Invalid_requestor! = Copy_add_invalid_requestor)

R3_2_Check_copy

⇧ R3_Add_copy (Copy?, Invalid_copy!)
Library_state (Ξ Library_copies)
Valid_copy_copy_add! : 𝔹

Copy? ➲ Valid_copy_copy_add! ⊗ Invalid_copy!
(Copy? ∉ Library_copies ∧ Valid_copy_copy_add! = True)
∨ (Copy? ∈ Library_copies ∧ Invalid_copy! = Copy_add_invalid_copy)

R3_3_Check_new_book

⇧ R3_Add_copy (Book?)
⇨ R3_1_Check_requestor (Valid_requestor_copy_add?)
⇨ R3_2_Check_copy (Valid_copy_copy_add?)
Library_state (Ξ Library_books)
New_book! : 𝔹
Not_new_book! : 𝔹

Book? ⍟ Valid_requestor_copy_add? ⍟ Valid_copy_copy_add? ➲
New_book! ⊗ Not_new_book!
(Book ∉ Library_books ∧ New_book! = True)
∨ (Book ∈ Library_books ∧ Not_new_book! = True)

R3_4_Record_new_book

⇧ R3_Add_copy (Copy?, Book?, Authors?, Subjects?)
⇨ R3_3_Check_new_book (New_book?)
Library_state (Δ Library_books, Δ Library_book_authors, Δ Library_book_subjects,
Δ Writes, ΔIs_a_subject_of)
Record_new_book_ok! : 𝔹

Copy? ⍟ Book ⍟ Authors ⍟ Subjects ⍟ New_book? ➲ Record_new_book_ok!
Library_books' = Library_books ∪ {Book?}
Library_book_authors' = Library_book_authors ∪ Authors?
Library_book_subjects' = Library_book_subjects ∪ Subjects?
Writes' = Writes ∪ Authors? ⤔ {Book?}
Is_a_subject_of' = Is_a_subject_of ∪ Subjects? ⤔ {Book?}
Record_new_book_ok! = True

R3_5_Record_new_copy

⇧ R3_Add_copy (Copy?, Book?, Valid_copy_add!)
⇨ R3_3_Check_new_book (Not_new_book?)
⇨ R3_4_Record_new_book (Record_new_book_ok?)
Library_state (Δ Available_copies, Δ Is_a_copy_of)

Copy?⨀Book? ⨀ (Not_new_book? ⊗ Record_new_book_ok?) ⊃
Valid_copy_add!
Available_copies' = Available_copies ∪ {Copy?}
Is_a_copy_of' = Is_a_copy_of ∪ {Copy? ↦ Book?}
Valid_copy_add! = Copy_add_ok

D. Remove a copy of a book from the library

The processes 4.1 to 4.5 (Figure 3-41) and the related RDSDs (Figures 3-42 to 3-46) can be translated into Z specifications as follows.

R4_1_Check_requestor

⇧ R4_Remove_copy (Requestor?, Invalid_requestor!)
Library_state (Ξ Staff)
Valid_requestor_copy_remove! : 𝔹

Requestor? ⊃ Valid_requestor_copy_remove! ⊗ Invalid_requestor!
(Requestor? ∈ Staff ∧ Valid_requestor_copy_remove! = True)
∨ (Requestor? ∉ Staff ∧ Invalid_requestor! = Copy_remove_invalid_requestor)

R4_2_Check_copy

⇧ R4_Remove_copy (Copy?, Invalid_copy!)
Library_state (Ξ Available_copies)
Valid_copy_copy_remove! : 𝔹

Copy? ⊃ Valid_copy_copy_remove! ⊗ Invalid_copy!
(Copy? ∈ Available_copies ∧ Valid_copy_copy_remove! = True)
∨ (Copy? ∉ Available_copies ∧ Invalid_copy! = Copy_remove_invalid_copy)

R4_3_Check_last_copy

⇧ R4_Remove_copy (Copy?)
⇨ R4_1_Check_requestor (Valid_requestor_copy_remove?)
⇨ R4_2_Check_copy (Valid_copy_copy_remove?)
Library_state (Ξ Is_a_copy_of)
Last_copy! : 𝔹
Not_last_copy! : 𝔹

Copy? ⨀ Valid_requestor_copy_remove? ⨀ Valid_copy_copy_remove? ⊃
Last_copy! ⊗ Not_last_copy!
((# Is_a_copy_of $^{-1}$(| Is_a_copy_of (| { Copy?} |) |) = 1) ∧ Last_copy! = True)
∨ ((# Is_a_copy_of $^{-1}$(| Is_a_copy_of (| { Copy?} |) |) > 1) ∧ Not_last_copy! = True)

R4_4_Record_book_remove

⇧ R4_Remove_copy (Copy?)
⇨ R4_3_Check_last_copy (Last_copy?)
Library_state (Δ Library_books. Δ Library_book_authors, Δ Library_book_subjects,
Δ Writes, Δ Is_a_subject_of)
Remove_book_ok!= 𝔹

Copy? ⍾ Last_copy? ⊃ Remove_book_ok!
Books = = Is_a_copy_of (| {Copy?} |)
Library_books ' = Library_books \ Books
Library_book_authors ' = Library_book_authors \
{ a : Writes $^{-1}$ (| Books |) | Write (| {a} |) = 1 }
Library_book_subjects ' = Library_book_subjects \
{ s : Is_a_subject_of $^{-1}$ (| Books |) | Is_a_subject_of (| {s} |) = 1 }
Writes ' = Writes ⩥ Books
Is_a_subject_of ' = Is_a_subject_of ⩥ Books
Remove_book_ok! = True

R4_5_Record_copy_remove

⇧ R4_Remove_copy (Copy?)
⇨ R4_3_Check_last_copy (Not_last_copy?)
⇨ R4_4_Record_book_remove (Remove_book_ok?)
Library_state (Δ Available_copies, Δ Is_a_copy_of)

Copy? ⍾ (Not_last_copy? ⊗ Remove_book_ok?) ⊃ Valid_copy_remove!
Available_copies ' = Available_copies \ {Copy?}
Is_a_copy_of ' = {Copy?} ⩤ Is_a_copy_of
Valid_copy_remove! = Copy_remove_ok

E. Enquiry

The processes 5.1 to 5.4 (Figure 3-47) can be translated into Z specifications as follows.

R5_1_Enquiry_by_author

⇧ R5_Enquiry (By_author_enquiry?, List_of_books_by_author!,
Invalid_by_author_enquiry!)

By_author_enquiry? ⊃ List_of_books_by_author! ⊗ Invalid_by_author_enquiry!

R5_2_Enquiry_by_subject

⇧ R5_Enquiry (By_subject_enquiry?, List_of_books_by_subject!,
Invalid_by_subject_enquiry!)

By_subject_enquiry? ⊃ List_of_books_by_subject! ⊗ Invalid_by_subject_enquiry!

R5_3_Enquiry_by_borrower

⇧ R5_Enquiry (By_borrower_enquiry?, List_of_copies_by_borrower!, Invalid_by_borrower_enquiry!)

By_borrower_enquiry? ➲ List_of_copies_by_borrower! ⊗ Invalid_by_borrower_enquiry!

R5_4_Enquiry_last_borrower

⇧ R5_Enquiry (Last_borrower_enquiry?, Last_borrower_info!, Invalid_last_borrower_enquiry!)

Last_borrower_enquiry? ➲ Last_borrower_info! ⊗ Invalid_last_borrower_enquiry!

E.1 Enquire the list of books by a particular borrower

The processes 5.1.1 to 5.1.3 (Figure 3-48) and the related RDSDs (Figures 3-49 to 3-50) can be translated into Z specifications as follows.

R5_1_1_Check_requestor

⇧ R5_1_Enquiry_by_author (Requestor?, Invalid_requestor!)
Library_state (Ξ Users)
Valid_requestor_by_author_enquiry! : 𝔹

Requestor? ➲ Valid_requestor_by_author_enquiry! ⊗ Invalid_requestor!
(Requestor? ∈ Users ∧ Valid_requestor_by_author_enquiry! = True)
∨ (Requestor? ∉ Users ∧ Invalid_requestor! = By_author_enquiry_invalid_requestor)

R5_1_2_Check_author

⇧ R5_1_Enquiry_by_author (Author?, Invalid_author!)
Library_state (Ξ Library_book_authors)
Valid_author! : 𝔹

By_author_enquiry? ➲ Valid_author! ⊗ Invalid_author!
(Author? ∈ Library_book_authors ∧ Valid_author! = True)
∨ (Author? ∉ Library_book_authors ∧
Invalid_author! = By_author_enquiry_invalid_author)

R5_1_3_Produce_books_by_author

⇧ R5_1_Enquiry_by_author (Author?, List_of_books_by_author!)
⇨ R5_1_1_Check_requestor (Valid_requestor_by_author_enquiry?)
⇨ R5_1_2_Check_author (Valid_author?)
Library_state (Ξ Writes)

Author? ⊙ Valid_requestor_by_author_enquiry? ⊙ Valid_author? ➲ List_of_books_by_author!
List_of_books_by_author! = Writes (| {Author?} |)

E.2 Enquire the list of books by a particular subject

The processes 5.2.1 to 5.2.3 (Figure 3-51) and the related RDSDs (Figures 3-52 to 3-53) can be translated into Z specifications as follows.

R5_2_1_Check_requestor

⇧ R5_1_Enquiry_by_subject (Requestor?, Invalid_requestor!)
Library_state (Ξ Users)
Valid_requestor_by_subject_enquiry! : 𝔹

By_subject_enquiry? ➲ Valid_requestor_by_subject_enquiry! ⊗ Invalid_requestor!
(Requestor? ∈ Users ∧ Valid_requestor_by_subject_enquiry! = True)
∨ (Requestor? ∉ Users ∧ Invalid_requestor! = By_subject_enquiry_invalid_requestor)

R5_2_2_Check_subject

⇧ R5_2_Enquiry_by_subject (Subject?, Invalid_subject!)
Library_state (Ξ Library_book_subjects)
Valid_subject! : 𝔹

Subject? ➲ Valid_subject! ⊗ Invalid_subject!
(Subject? ∈ Library_book_subjects ∧ Valid_subject! = True)
∨ (Subject? ∉ Library_book_subjects ∧
Invalid_subject! = By_subject_enquiry_invalid_subject)

R5_2_3_Produce_books_by_subject

⇧ R5_2_Enquiry_by_subject (Subject?, List_of_books_by_subject!)
⇨ R5_2_1_Check_requestor (Valid_requestor_by_subject_enquiry?)
⇨ R5_2_2_Check_subject (Valid_subject?)
Library_state (Ξ Is_a_subject_of)

Subject? ⊙ Valid_requestor_by_subject_enquiry? ⊙ Valid_subject? ➲
List_of_books_by_suject!
List_of_books_by_subject! = Is_a_subject_of (| {Subject?} |)

E.3 Enquire the list of copies checked out by a particular borrower

The processes 5.3.1 to 5.3.5 (Figure 3-54) and the related RDSDs (Figures 3-55 to 3-59) can be translated into Z specifications as follows.

R5_3_1_Check_user

⇧ R5_3_Enquiry_by_borrower (Requestor?, Invalid_user!)
Library_state (Ξ Users)
Valid_user! : 𝔹

Requestor? ➲ Valid_user! ⊗ Invalid_user!
(Requestor? ∈ Users ∧ Valid_user! = True)
∨ (Requestor? ∉ Users ∧ Invalid_user! = By_borrower_enquiry_invalid_user)

R5_3_2_Check_staff

⇧ R5_3_Enquiry_by_borrower (Requestor?)
⇨ R5_3_1_Check_user (Valid_user?)
Library_state (Ξ Staff)
Valid_staff! : 𝔹
Not_a_staff! : 𝔹

Requestor? ⊙ Valid_user ➲ Valid_staff! ⊗ Not_a_staff!
(Requestor? ∈ Staff ∧ Valid_staff! = True)
∨ (Requestor? ∉ Staff ∧ Not_a_staff! = True)

R5_3_3_Check_authorized_requestor

⇧ R5_3_Enquiry_by_borrower (Requestor?, Borrower?, Unauthorized_requestor!)
⇨ R5_3_2_Check_staff (Not_a_staff?)
Authorized_requestor! : 𝔹

Requestor? ⊙ Borrower? ⊙ Not_a_staff? ➲
Authorized_requestor! ⊗ Unauthorized_requestor!
(Requestor? = Borrower? ∧ Authorized_requestor! = True)
∨ (Requestor? ≠ Borrower? ∧
Unauthorized_requestor! = By_borrower_enquiry_unauthorized_requestor)

R5_3_4_Check_borrower

⇧ R5_3_Enquiry_by_borrower (Borrower?, Invalid_borrower!)
Library_state (Ξ Borrowers)
Valid_borrower_by_borrower_enquiry! : 𝔹

Borrower? ➲ Valid_borrower_by_borrower_enquiry! ⊗ Invalid_borrower!
(Borrower? ∈ Borrowers ∧ Valid_borrower_by_borrower_enquiry! = True)
∨ (Borrower? ∉ Borrowers ∧
Invalid_borrower! = By_borrower_enquiry_invalid_borrower)

R5_3_5_Produce_copies_by_borrower

⇧ R5_3_Enquiry_by_borrower (Borrower?, List_of_copies_by_borrower!)
⇨ R5_3_2_Check_staff (Valid_staff)
⇨ R5_3_3_Check_authorized_requestor (Authorized_requestor?)
⇨ R5_3_4_Check_borrower (Valid_borrower_by_borrower_enquiry?)
Library_state (Ξ Currently_checks_out)

Borrower? ⍟ (Valid_staff ⊗ Authorized_requestor?) ⍟
Valid_borrower_by_borrower_enquiry? ⊃ List_of_copies_by_borrower!
List_of_copiess_by_borrower! = Currently_checks_out (| {Borrower?} |)

E.4 Enquire what borrower last checked out a particular copy a book

The processes 5.4.1 to 5.4.3 (Figure 3-60) and the related RDSDs (Figures 3-61 to 3-62) can be translated into Z specifications as follows.

R5_4_1_Check_requestor

⇧ R5_4_Enquiry_last_borrower (Requestor?, Invalid_requestor!)
Library_state (Ξ Staff)
Valid_requestor_last_borrower_enquiry! : 𝔹

Requestor? ⊃ Valid_requestor_last_borrower_enquiry! ⊗ Invalid_requestor!
(Requestor? ∈ Staff ∧ Valid_requestor_last_borrower_enquiry! = True)
∨ (Requestor? ∉ Staff ∧
Invalid_requestor! = Last_borrower_enquiry_invalid_requestor)

R5_4_2_Check_copy

⇧ R5_4_Enquiry_last_borrower (Copy?, Invalid_copy!)
Library_state (Ξ Checked_out_copies)
Valid_copy_last_borrower_enquiry! : 𝔹

Copy? ⊃ Valid_copy_last_borrower_enquiry! ⊗ Invalid_copy!
(Copy? ∈ Checked_out_copies ∧ Valid_copy_last_borrower_enquiry! = True)
∨ (Copy? ∉ Checked_out_copies ∧
Invalid_copy! = Last_borrower_enquiry_invalid_copy)

R5_4_3_Produce_last_borrower_info

⇧ R5_4_Enquiry_last_borrower (Copy?, Last_borrower_info!)
⇨ R5_4_1_Check_requestor (Valid_requestor_last_borrower_enquiry?)
⇨ R5_4_2_Check_copy (Valid_copy_last_borrower_enquiry?)
Library_state (Ξ Currently_checks_out)

Copy? ⍟ Valid_requestor_last_borrower_enquiry? ⍟
Valid_copy_last_borrower_enquiry? ⊃ Last_borrower_info!
Last_borrower_info! = Currently_checks_out $^{-1}$ (Copy?)

Chapter 4 Software requirements verification technique

4.1 Overview

In this chapter, a technique for verifying software requirements is proposed. A software requirements specification produced must be verified (informally and formally proved) against the end-users' actual requirements and also it must be verified that it is internally consistent.

In section 4.2, the overview of the proposed software requirements verification technique is given. This includes the steps to be carried out. Then the details of each step are described in sections 4.3, 4.4, and 4.5. Both semiformal and formal requirements specifications must be verified. Section 4.3 explains how to check (informally proved) the consistency within the semiformal requirements specification itself. Section 4.4 discusses how to check (informally prove) the consistency of the formal requirements specification against the semiformal requirements specification. Finally, section 4.5 demonstrates how to apply formal proofs to verify that the formal requirements specification is internally consistent.

4.2 Overview of the proposed software requirements verification technique

Software requirements verification is the process of evaluating that the software requirements specification satisfies the end-users' actual requirements. This includes:

1) Proving that the software requirements specification is a complete and correct expression of the end-users' requirements.
2) Proving that the software requirements specification is internally consistent.

Software errors categories, for example those given in section 1.7, are very useful in verifying software requirements.

There are various techniques that contribute to software requirements verification. The techniques range from informal proof techniques to the other end of formal proof techniques. The informal proof techniques include for example informal checking, reviews, walkthrough, and inspection [31, 94], and also animation or prototyping. Formal proof techniques include all techniques that are based on mathematical proofs, and also symbolic execution techniques. All of them are aimed at finding errors in software requirements specifications.

The software requirements verification technique proposed in this thesis uses both informal and formal proof techniques.

Figure 4-1 shows the scope of the proposed technique (bold lines represent the areas of application of the proposed technique).

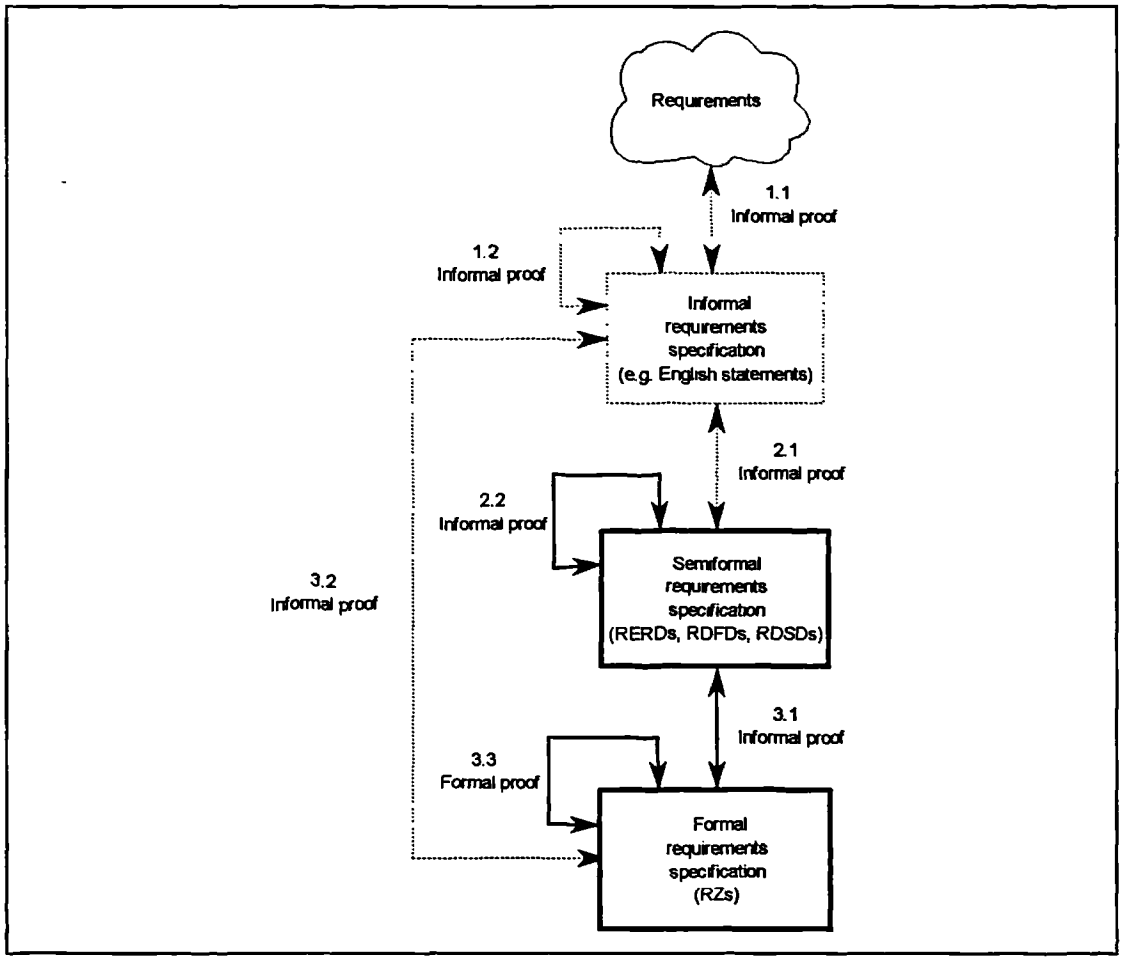


Figure 4-1: Software requirements verification (bold lines represent areas of application of the proposed technique)

The informal and formal proofs shown in Figure 4-1 are briefly explained as follows:

Step 1.1

Informal proof is applied to check the consistency, correctness, and completeness of the informal requirements specification against the end-users' requirements.

Step 1.2

Informal proof is applied to check the consistency within the informal requirements specification itself.

Step 2.1

Informal proof is applied to check the consistency, correctness, and completeness of the semiformal requirements specification (RERDs, RDFDs, and RDSDs) against the informal requirements specification.

Step 2.2

Informal proof is applied to check the consistency within the semiformal requirements specification (RERDs, RDFDs, and RDSDs).

Step 3.1

Informal proof is applied to check the consistency, correctness, and completeness of the formal requirements specification (RZs) against the semiformal requirements specification (RERDs, RDFDs, and RDSDs).

Step 3.2

Informal proof is applied to check the consistency, correctness, and completeness of the formal requirements specification (RZs) against the informal requirements specification.

Step 3.3

Formal proof is applied to mathematically prove the consistency within the formal requirements specification (RZs).

The informal requirements specification must be verified against the end-users' requirements (step 1.1) as well as it must be verified that the statements stated therein do not conflict or are not inconsistent (step 1.2).

According to chapter 2, the semiformal requirements specification is developed from the informal requirements specification, therefore the former is verified against the latter (step 2.1). In addition, the former must be verified that it contains no conflicts or any inconsistency (step 2.2).

Similarly, the formal requirements specification is developed from both the semiformal requirements specification and the informal requirements specification, therefore informal and formal proofs are required as stated in steps 3.1, 3.2, and 3.3 above.

The proposed software requirements verification technique covers only steps 2.2, 3.1, and 3.3. The proposed technique can be applied by using the following steps.

4.3 Step 2.2: informally prove the semiformal requirements specification

Step 2.2 is concerned with informally proving (or checking) the consistency within the semiformal requirements specification itself. This includes checking the consistency within each of the three diagrams, the RERDs, RDFDs, and RDSDs, and checking the consistency of the three diagrams against each other.

The tasks in step 2.2 are:

1) Check the RERDs
2) Check the RDFDs
3) Check the RDSDs
4) Check the RERDs against the RDSDs
5) Check the RDFDs against the RDSDs
6) Check the RDFDs against the RERDs

4.3.1 Check the RERDs

The RERDs drawn must satisfy these syntax and semantics:

- Each entity box must have an entity name and an entity type, and the entity name must be unique.
- The entity must participate in at least one relationship.
- Each relationship must be given a name and if a relationship is an interrogative relationship, the name must be unique.
- For the subset relationship ("Is_a" relationship), the cardinality must be one-to-one, and the subsets must be conjoined via either an exclusive or inclusive conjunction notation.

4.3.2 Check the RDFDs

The RDFDs drawn must satisfy these syntax and semantics:

- All external entities, processes, data flows (except data flows from/to data stores), and data stores must be labelled.
- External entities must be present only on the context diagram.
- The context diagram must contain only one process and at least one external entity.
- A data flow can flow between an external entity and a process, two processes, or a process and a data store. In other words, a data flow cannot flow between two external entities, two data stores, or an external entity and a data store.
- Each process must have at least one input data flow and one output data flow.
- Each data store must have at least one input or one output data flow from or to a process. If a data flow flows from a data store to a process, the data store access must be Ⓡ; otherwise it may be either Ⓒ, Ⓤ, or Ⓓ.
- Input and output data flows of the parent and its sub-processes must be consistent.

4.3.3 Check the RDSDs

The RDSDs drawn must satisfy these syntax and semantics:

- Each box must have a data item name.
- Data item names on the same RDSD must be unique.
- Each primitive data item must be given a data type.
- If an RDSD contains more than one data components, they must be connected via one or more data interface notations: conjunction, disjunction, and exclusive disjunction.
- Data components of an RDSD may be defined on other RDSDs, but the definition must not be recursive.

4.3.4 Check the RERDs against the RDSDs

The rule for balancing the RERDs against the RDSDs is as follow:

- Each entity on the RERDs may or may not have a corresponding RDSD. An RDSD is drawn only if the specifier decided to define the attributes of the entity type.

4.3.5 Check the RDFDs against the RDSDs

The rule for balancing the RDFDs against the RDSDs is as follow:

- Each data flow on the RDFD must appear on the RDSD.

4.3.6 Check the RDFDs against the RERDs

The rule for balancing the RDFDs against the RERDs is as follow:

- The data stores on the RDFDs are the entities and interrogative relationships on the RERDs.

4.4 Step 3.1: informally prove the formal requirements specification against the semiformal requirements specification

Step 3.1 is concerned with informally proving (or checking) the consistency of the formal requirements specification against the semiformal requirements specification. In other words, step 3.1 is concerned with checking the consistency of the RZs against the RERDs, RDFDs, and RDSDs.

The tasks in step 3.1 are:

1) Check the RZ state specifications against the RERDs and RDSDs
2) Check the RZ operation specifications against the RDFDs and RDSDs

4.4.1 Check the RZ state specifications against the RERDs and RDSDs

The rules for balancing the RZ state specifications against the RERDs and RDSDs are as follows:

- If the data type of the entity on the RERDs is further defined on the RDSD, that data type must be defined as a schema type in the RZ state specifications, otherwise it must be defined as a basic type or free type.
- Each entity on the RERDs must be defined in the RZ state specifications.
- Each interrogative relationships on the RERDs must be defined as a relationship (see section 2.5.1.1) in the RZ state specifications.
- The constraints among subsets of the same superset as shown on the RERDs must correspond with the predicate(s) in the RZ state specifications.

4.4.2 Check the RZ operation specifications against the RDFDs and RDSDs

The rule for balancing the RZ operation specifications against the RDFDs and RDSDs is as follow:

- Each process on the RDFDs must have a corresponding schema in the RZ operation specifications.

4.5 Step 3.3: formally prove the formal requirements specification

Step 3.3 is concerned with formally proving the consistency within the formal requirements specification (RZs). The classification of software errors given in section 1.7 is used as a guideline for the proofs described here. The rest of this chapter describes how to apply formal proofs to verify that the formal requirements specification is internally consistent. The proofs are carried out to find out whether there is any inconsistency (as stated in section 1.7) in the formal requirements specification; or in other words to prove that there is no inconsistency in the formal requirements specification.

4.5.1 Inconsistent critical requirements

Critical requirements or invariant requirements of a system are requirements that must hold in every state. The critical requirements may be thought of as preconditions to the entire specification, rather than as being specific to any particular operation; they are preconditions of every operation defined on the system.

The critical requirements are defined in one or more state schemas. The critical requirements defined in the RZ state specifications may be in conflict or inconsistent, therefore formal proofs are required in order to make sure that there is no inconsistency among those critical requirements. Formal proofs of the consistency of the critical requirements can be carried out independently for each state schema.

According to the RZ state specifications of the library system, the proofs can be carried out in five sub-proofs, one for each state schema namely "Person_state", "Author_state", "Subject_state", Copy_state", and "Library_state".

If there is only one critical requirement defined in that schema, there shall be no inconsistency. If there is more than one critical requirement, formal proof is required.

To prove that there is no inconsistent critical requirements, these steps are suggested:

1) First, for each state schema, the critical requirements are extracted from that state schema. All predicates in the state schema are critical requirements. Critical requirements are also sometimes expressed in the declaration part of the schema as well [86].

   The schema "Person_state" has only one critical requirement as follow:

   <Borrowers, Staff> partition Users

   The schemas "Author_state", "Subject_state", and "Book_state" have no critical requirement.

The schema "Copy_state" has only one critical requirement as follow:

<Checked_out_copies, Available_copies> partition Library_copies

The schema "Library_state" has twelve critical requirements (even though the schema "Library_state" includes the four schemas above, they are not considered because they are proved separately). The critical requirements of the schema "Library_state" are:

dom Writes = Library_book_authors
ran Writes = Library_books
dom Is_a_subject_of = Library_book_subjects
ran Is_a_subject_of = Library_books
dom Is_a_copy_of = Library_copies
ran Is_a_copy_of = Library_books
$\forall c$: Library_copies • # Is_a_copy_of (| {c} |) = 1
dom Currently_checks_out $\subseteq$ Borrowers
ran Currently_checks_out = Checked_out_copies
$\forall c$: Checked_out_copies • # Currently_checks_out $^{-1}$ (| {c} |) = 1
Max_copy > 0
$\forall b$: dom Currently_checks_out • # Currently_checks_out (| {b} |) $\leq$ Max_copy

2) Then, formal proofs are carried out for each schema. The critical requirements of the library system must be consistent.

However, in case that the critical requirements are inconsistent, formal proofs can reveal them as shown as follows.

Assume that the critical requirements of the system are defined as follows:

$B \cap C = \varnothing \wedge$
$B \cup C = A \wedge$
$D \cup C = B \wedge$
$C \neq \varnothing$

From the critical requirements above, we then can derive:

1. $B \cap C = \varnothing$	premise
2. $D \cup C = B$	premise
3. $(D \cup C) \cap C = \varnothing$	1,2 substitution
4. $C = \varnothing$	3

Therefore, the proof shows that there is inconsistency ($C \neq \varnothing$ and $C = \varnothing$ which is impossible) within those critical requirements.

4.5.2 Inconsistency between the state specification and the initial state specification

Both the state space specification and the initial state specification are defined. However, there must be at least one initial state that satisfies the state space specification. Formal proof of the consistency between the state specification and the initial state can be carried out, see[86] for more details.

4.5.3 Inconsistent interface specifications

According to the concept of the data flow diagrams, the process can be decomposed into a network of sub-processes and the input and output data flows of the parent and its sub-processes must be consistent [53].

The interface specifications are specified in RZ operation specifications by using the data interface symbols (see section 2.5.1.3) and the input/output data flow relation symbol (see section 2.5.1.5).

In this section, the proof concerning the consistency of the interface specifications (the consistency of the process decomposition) is given.

The proof method is suggested by Kung in [53]. However, the proof offered in this thesis is quite different from the one given in [53]. The proof offered in this thesis is a lot more complete than the one in [53].

The proof proposed in this thesis is explained as follows.

From the Z specifications of the library system, which are given in section 3.5, the interface specifications of the process 1 "R1_Check_out_copy" and its sub-processes are as follows.

From the schema "R1_Check_out_copy", the interface specification is defined as:

Copy_check_out? ➲ Valid_copy_check_out! ⊗ Invalid_copy_check_out!

The process "R_1_Check_out_copy" is decomposed into two sub-processes namely "R1_1_Check_copy_check_out" and "R1_2_Record_copy_check_out".

The interface specification of the process "R1_1_Check_copy_check_out" is:

Copy_check_out? ➲ Valid_copy_check_out_check! ⊗ Invalid__copy_check_out! (1)

The interface specification of the process "R1_2_Record_copy_check_out" is:

Borrower? Ⓐ Copy? Ⓐ Valid_copy_check_out_check? ➲ Valid_copy_check_out! (2)

The proof is required to prove that given the input data flow of the parent process ("R1_Check_out_copy"), its sub-processes ("R1_1_Check_copy_check_out" and "R1_2_Record_copy_check_out") must produce the same output data flows as the parent process. In other words, given the input data flow "Copy_check_out", the proof must show that the processes "R1_1_Check_copy_check_out" and "R1_2_Record_copy_check_out" produce the output data flows "Valid_copy_check_out" and "Invalid_copy_check_out" and the proof must also show that either of the two output data flows is produced but not both.

The proof is carried out by using *non-logical rules* and *logical rules*.

The non-logical rules are the application dependent rules. The statements (1) and (2) above are non-logical rules.

The logical rules are logically valid and can be applied to all applications. There are 20 logical rules, L1 to L20, as proposed in this thesis. The logical rules are derived by considering the different basic elements that can occur in combination in DFDs. The first three rules, L1 to L3, show how individual inputs and outputs can be related. The next 13 rules, L4 to L16, illustrate the behaviour of different possible DFD fragments. The last four rules, L17 to L20, correspond to the basic associative properties.

L1. a? Ⓐ b? ⇒ a?, a? Ⓐ b? ⇒ b?

L1 states that for any process P, if a? and b? are inputs of P, then only a? or only b? can be introduced as the input of the sub-process.

In other words, all inputs of the parent process are entitled to be the inputs of its sub-processes; however each sub-process may not require all the inputs provided by the parent process. Therefore, this rule allows only some inputs to be extracted from all inputs provided.

L2. a?, b! ⇒ a? Ⓐ b!

L2 states that if a? is the input of the parent process and b! is the output derived from the proof, then a? can be introduced and combined with the output b!.

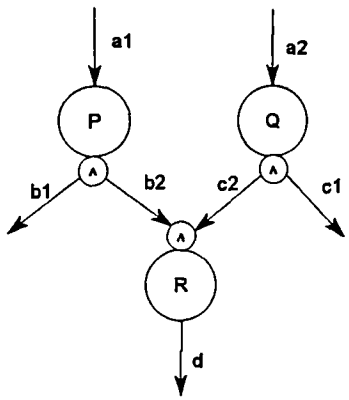
L3. a? Ⓐ b! ⇒ a? Ⓐ b?

L3 states that the output of one sub-process can be passed as the input to another sub-process on the same DFD.

L4. b1! Ⓐ b2!, c1! Ⓐ c2! ⇒ b1! Ⓐ c1! Ⓐ (b2? Ⓐ c2?)

L4 states that for any processes P and Q, if P produces b1! Ⓐ b2! and Q produces c1! Ⓐ c2!, then we can derive b1! Ⓐ c1! Ⓐ (b2? Ⓐ c2?).

This rule can be shown by the DFD as follows.



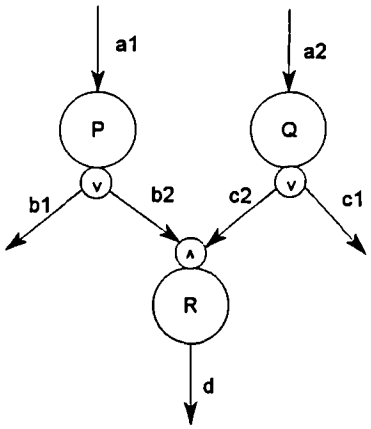
Given the process P which produces the outputs b1! Ⓐ b2! and the process Q which produces the outputs c1! Ⓐ c2!, if the outputs b2! and c2! are passed as the inputs to the process R and the two outputs are in conjunction, we then can derive b1! Ⓐ c1! Ⓐ (b2? Ⓐ c2?). This is true because from the diagram above, it shows that all of these three outputs b1!, c1!, and d! are produced (b1! Ⓐ c1! Ⓐ d!). Therefore b1! Ⓐ c1! Ⓐ (b2? Ⓐ c2?) is a valid consequence because whatever output(s) produced by the inputs (b2? Ⓐ c2?) they must be in conjunction with the outputs b1! and c1!.

Rules L5 to L16 can be reasoned in a similar way.

L5. b1! Ⓥ b2!, c1! Ⓥ c2! ⇒ b1! Ⓥ c1! Ⓥ (b2? Ⓐ c2?)

L5 states that for any processes P and Q, if P can produce b1! Ⓥ b2! and Q can produce c1! Ⓥ c2!, then we can derive b1! Ⓥ c1! Ⓥ (b2? Ⓐ c2?).

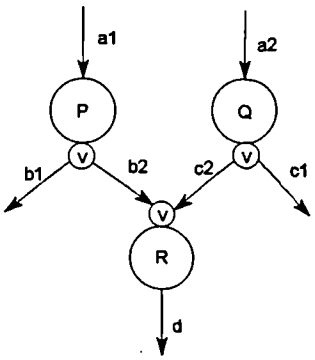
This rule can be shown by the DFD as follows.



L6. b1! ⓥ b2!, c1! ⓥ c2! ⇒ (b1! ⓐ c1!) ⓧ (b2? ⓥ c2?)ⓧ
((b1! ⓥ c1!) ⓐ (b2? ⓥ c2?))

L6 states that for any processes P and Q, if P produces b1! ⓥ b2! and Q produces c1! ⓥ c2!, then we can derive (b1! ⓐ c1!) ⓧ (b2? ⓥ c2?)ⓧ ((b1! ⓥ c1!) ⓐ (b2? ⓥ c2?)).

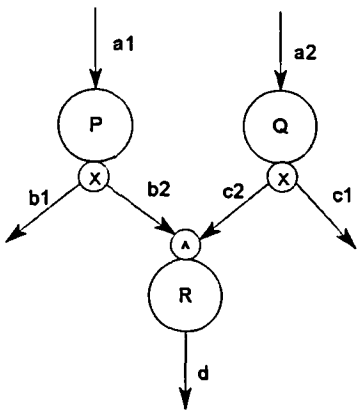
This rule can be shown by the DFD as follows.



L7. b1! ⓧ b2!, c1! ⓧ c2! ⇒ (b1! ⓥ c1!) ⓧ (b2? ⓐ c2?)

L7 states that for any processes P and Q, if P produces b1! ⓧ b2! and Q produces c1! ⓧ c2!, then we can derive (b1! ⓥ c1!) ⓧ (b2? ⓐ c2?).

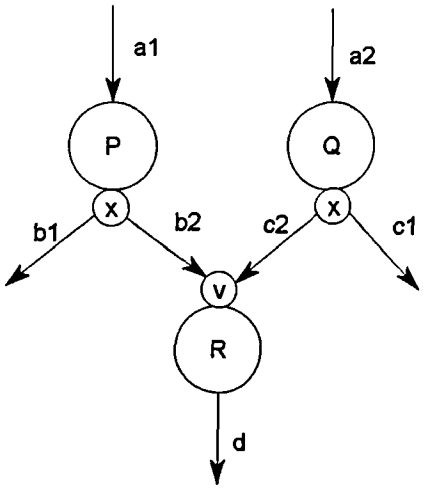
This rule can be shown by the DFD as follows.



L8. $b1! \otimes b2!, c1! \otimes c2! \Rightarrow (b1! \wedge c1!) \otimes (b2? \vee c2?) \otimes ((b1! \otimes c1!) \wedge (b2? \vee c2?))$

L8 states that for any processes P and Q, if P produces b1! ⊗ b2! and Q produces c1! ⊗ c2!, then we can derive (b1! ∧ c1!) ⊗ (b2? ∨ c2?) ⊗ ((b1! ⊗ c1!) ∧ (b2? ∨ c2?)).

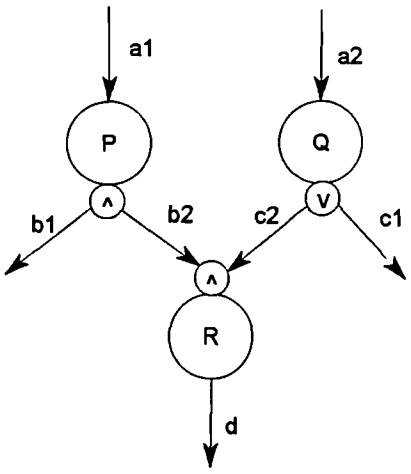
This rule can be shown by the DFD as follows.



L9. $b1! \wedge b2!, c1! \vee c2! \Rightarrow b1! \wedge (c1! \vee (b2? \wedge c2?))$

L9 states that for any processes P and Q, if P produces b1! ∧ b2! and Q produces c1! ∨ c2!, then we can derive b1! ∧ (c1! ∨ (b2? ∧ c2?)).

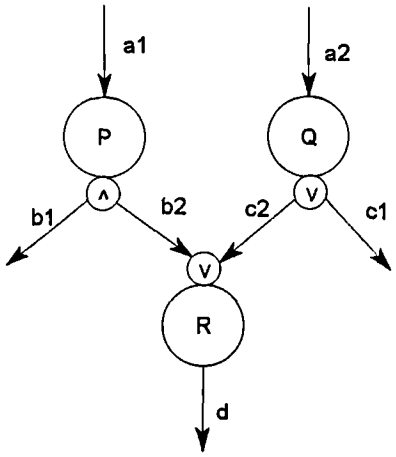
This rule can be shown by the DFD as follows.



L10. $b1! \wedge b2!, c1! \vee c2! \Rightarrow (b1! \wedge (b2? \vee c2?)) \otimes (b1! \wedge c1! \wedge (b2? \vee c2?))$

L10 states that for any processes P and Q, if P produces b1! ∧ b2! and Q produces c1! ∨ c2!, then we can derive (b1! ∧ (b2? ∨ c2?)) ⊗ (b1! ∧ c1! ∧ (b2? ∨ c2?)).

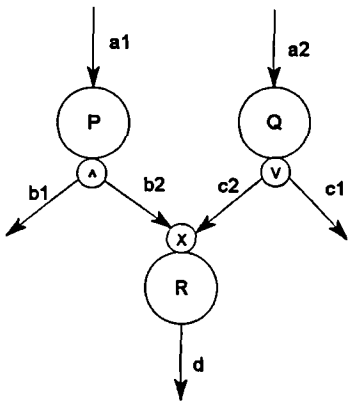
This rule can be shown by the DFD as follows.



L11. $b1! \otimes b2!, c1! \oplus c2! \Rightarrow b1! \otimes (b1! \otimes c1!) \otimes (b1! \otimes c1! \otimes (b2? \otimes c2?))$

L11 states that for any processes P and Q, if P produces $b1! \otimes b2!$ and Q produces $c1! \oplus c2!$, then we can derive $b1! \otimes (b1! \otimes c1!) \otimes (b1! \otimes c1! \otimes (b2? \otimes c2?))$.

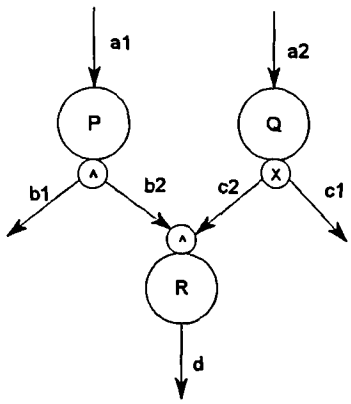
This rule can be shown by the DFD as follows.



L12. $b1! \otimes b2!, c1! \otimes c2! \Rightarrow b1! \otimes (c1! \otimes (b2? \otimes c2?))$

L12 states that for any processes P and Q, if P produces $b1! \otimes b2!$ and Q produces $c1! \otimes c2!$, then we can derive $b1! \otimes (c1! \otimes (b2? \otimes c2?))$.

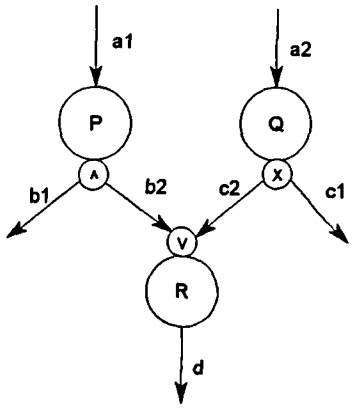
This rule can be shown by the DFD as follows.



L13. b1! Ⓐ b2!, c1! Ⓧ c2! ⇒ (b1! Ⓐ (b2? Ⓥ c2?)) Ⓧ
(b1!Ⓐ c1!Ⓐ (b2? Ⓥ c2?))

L13 states that for any processes P and Q, if P produces b1! Ⓐ b2! and Q produces c1! Ⓧ c2!, then we can derive (b1! Ⓐ (b2? Ⓥ c2?)) Ⓧ (b1!Ⓐ c1! Ⓐ (b2? Ⓥ c2?)).

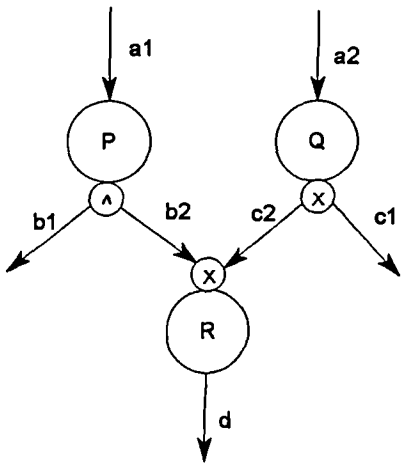
This rule can be shown by the DFD as follows.



L14. b1! Ⓐ b2!, c1! Ⓧ c2! ⇒ b1! Ⓧ (b1!Ⓐ c1! Ⓐ (b2? Ⓧ c2?))

L14 states that for any processes P and Q, if P produces b1! Ⓐ b2! and Q produces c1! Ⓧ c2!, then we can derive b1! Ⓧ (b1!Ⓐ c1! Ⓐ (b2? Ⓧ c2?)).

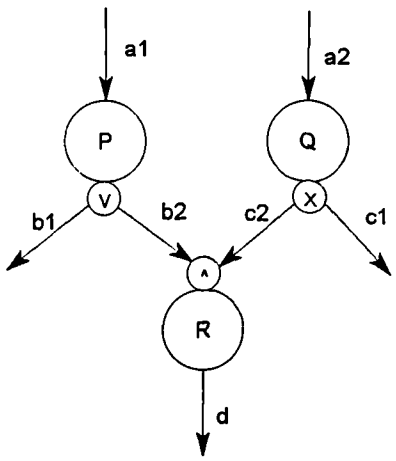
This rule can be shown by the DFD as follows.



L15. b1! Ⓥ b2!, c1! Ⓧ c2! ⇒ (b1!Ⓥ c1!) Ⓧ (b2? Ⓐ c2?) Ⓧ
(b1! Ⓐ (b2? Ⓐ c2?))

L14 states that for any processes P and Q, if P produces b1! Ⓥ b2! and Q produces c1! Ⓧ c2!, then we can derive (b1!Ⓥ c1!) Ⓧ (b2? Ⓐ c2?) Ⓧ (b1! Ⓐ (b2? Ⓐ c2?)).

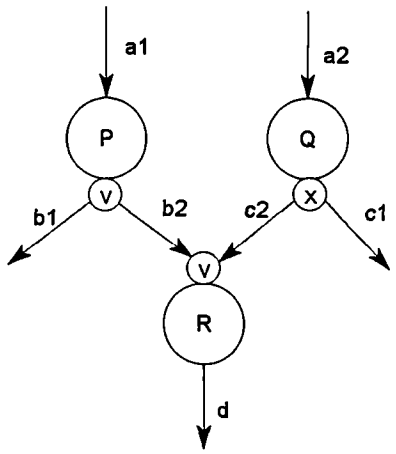
This rule can be shown by the DFD as follows.



L16. b1! Ⓥ b2!, c1! Ⓧ c2! ⇒ (b1!Ⓐ c1!) Ⓧ (b2? Ⓥ c2?) Ⓧ
((b1! Ⓥ c1!)Ⓐ (b2? Ⓥ c2?))

L16 states that for any processes P and Q, if P produces b1! Ⓥ b2! and Q produces c1! Ⓧ c2!, then we can derive (b1!Ⓐ c1!) Ⓧ (b2? Ⓥ c2?) Ⓧ ((b1! Ⓥ c1!)Ⓐ (b2? Ⓥ c2?)).

This rule can be shown by the DFD as follows.



L17. a! Ⓐ (b! Ⓐ c!) ⇔ (a! Ⓐ b!)Ⓐ c!

L17 states that if a! Ⓐ (b! Ⓐ c!) can be produced, then (a! Ⓐ b!)Ⓐ c! can also be produced and vice versa.

L18. a! Ⓥ (b! Ⓥ c!) ⇔ (a! Ⓥ b!)Ⓥ c!

L18 states that if a! Ⓥ (b! Ⓥ c!) can be produced, then (a! Ⓥ b!)Ⓥ c! can also be produced and vice versa.

L19. a! Ⓧ (b! Ⓧ c!) ⇔ (a! Ⓧ b!)Ⓧ c!

L19 states that if a! Ⓧ (b! Ⓧ c!) can be produced, then (a! Ⓧ b!)Ⓧ c! can also be produced and vice versa.

L20. a? Ⓐ (b! Ⓧ c!) ⇔ (a? Ⓐ b!)Ⓧ c!

L20 states that if a? Ⓐ (b! Ⓧ c!) can be produced, then (a? Ⓐ b!)Ⓧ c! can also be produced and vice versa.

The proof starts with the input data flows of the parent process. Then nonlogical and logical rules are applied until the result (the output data flows of the parent process) is obtained. In other words, given the input data flows of the parent process, we must prove that its sub-processes produce the same output data flows as the parent process.

From the library system, we will show how to prove the consistency in process decomposition of the process 1 "Check_out_copy". The RDFD of the process 1 is shown in Figure 4-2 and the RDFD of its sub-processes is shown in Figure 4-3.

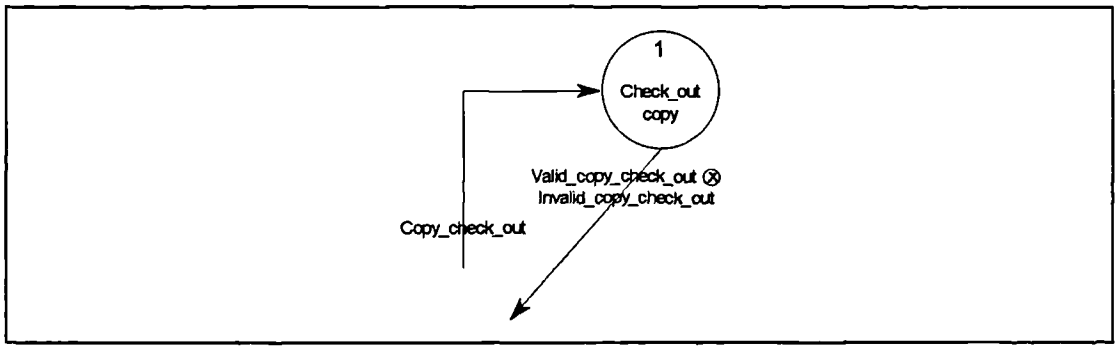


Figure 4-2: The RDFD of the process 1 "Copy_check_out"

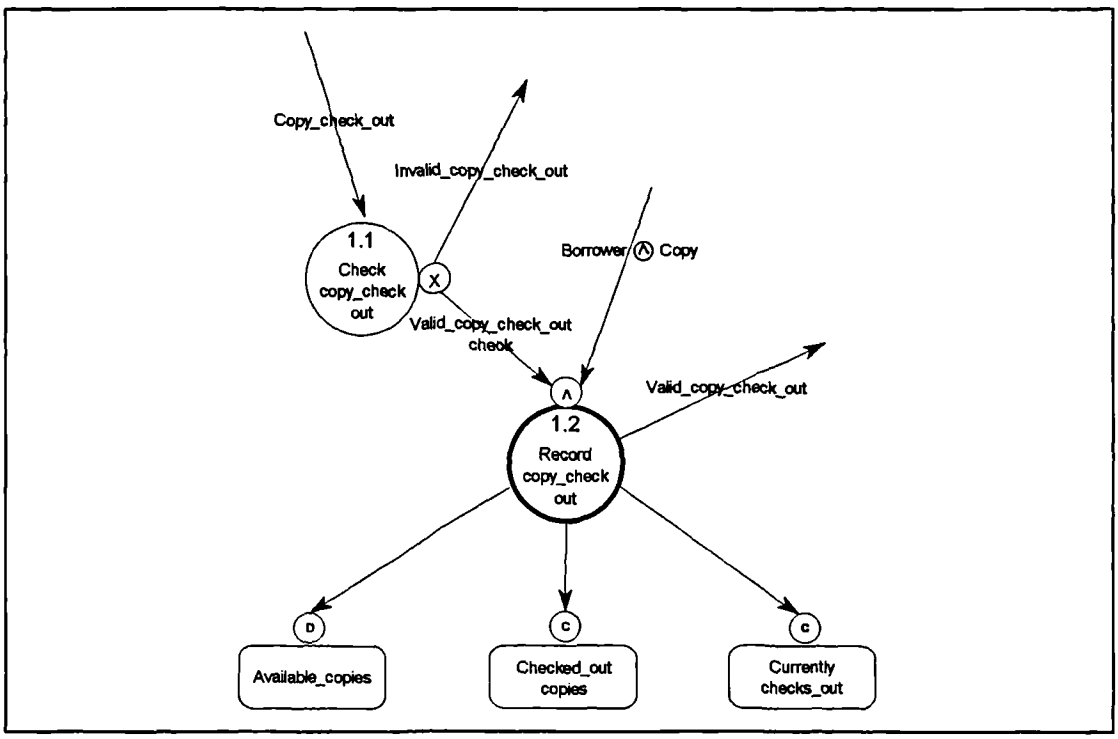


Figure 4-3: The decomposition of the process 1 RDFD

There are two nonlogical rules (which are defined in the Z specifications in the schemas "R1_1_Check_copy_check_out" and "R1_2_Record_copy_check_out") as follows:

NL1 Copy_check_out? ⊃ Valid_copy_check_out_check! ⊗ Invalid__copy_check_out!
NL2 Borrower? ∧ Copy? ∧ Valid_copy_check_out_check? ⊃ Valid_copy_check_out!

We must prove that given the input data flow "Copy_check_out?" (which is the input data flow of the parent process "R1_Check_out_copy"), the output data flows "Valid_copy_check_out! ⊗ Invalid_copy_check_out!" can be produced.

Copy_check_out? ➲ Valid_copy_check_out! ⊗ Invalid_copy_check_out!

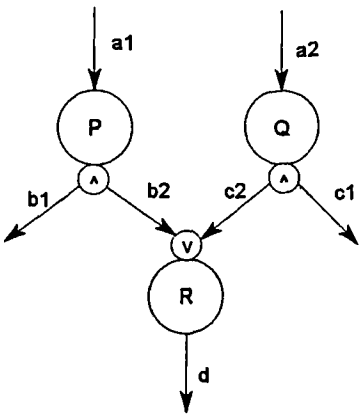
The proof is shown as follows.

1) Copy_check_out?	Given input
2) Valid_copy_check_out_check! ⊗ Invalid__copy_check_out!	NL1, 1)
3) Borrower? ⍓ Copy?	L1, 1)
4) (Borrower? ⍓ Copy?) ⍓ (Valid_copy_check_out_check! ⊗ Invalid__copy_check_out!)	L2, 2), 3)
5) (Borrower? ⍓ Copy? ⍓ Valid_copy_check_out_check!) ⊗ Invalid_copy_check_out!	L20, 4)
6) (Borrower? ⍓ Copy? ⍓ Valid_copy_check_out_check?) ⊗ Invalid_copy_check_out!	L3, 5)
7) Valid_copy_check_out! ⊗ Invalid_copy_check_out!	NL2, 6)

The rules L4 to L16 above are derived from 13 data flow diagrams which have valid data interfaces. There are other 5 data flow diagrams which can be drawn to depict all the combinations of the data interface notations. However, such 5 data flow diagrams are invalid. This is why there are no logical rules for these 5 invalid data flow diagrams.

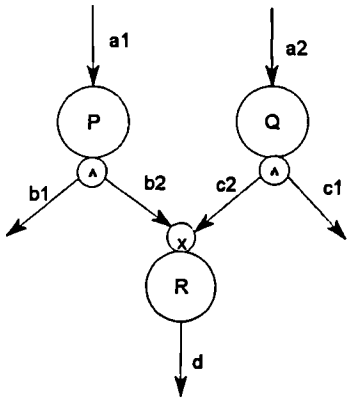
The 5 invalid data flow diagrams mentioned above are:

1)



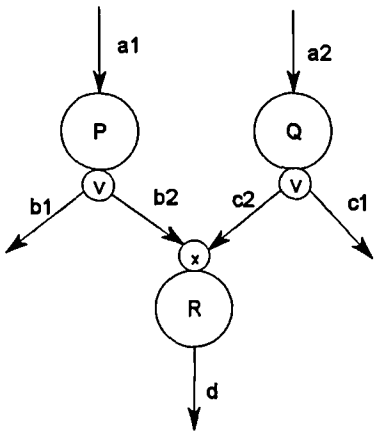
This data flow diagram is invalid because the data flows b2 and c2 must be in conjunction when they are passed as the input data flows into the process R. If the process P always produces the output data flow b2 (as shown on the diagram above, the process P always produces both the output data flows b1 and b2) and the process Q always produces the output data flow c2 (as shown on the diagram above, the process Q always produces both the output data flows c1 and c2), there will be no case that either one of them will be produced and then sent as the input data flow into the process R; therefore the data flows b2 and c2 must be in conjunction when they are sent as the input data flows into the process R. However, as shown on the diagram above, the data flows b2 and c2 are in disjunction when they are passed as the input data flows into the process R, therefore this data flow diagram is invalid.

2)



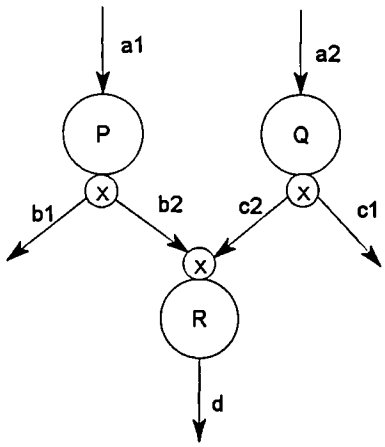
As already discussed in 1), the data flows b2 and c2 must be in conjunction when they are passed as the input data flows into the process R. As shown on the diagram above, the data flows b2 and c2 are in exclusive disjunction when they are passed as the input data flows into the process R, therefore this data flow diagram is invalid.

3)



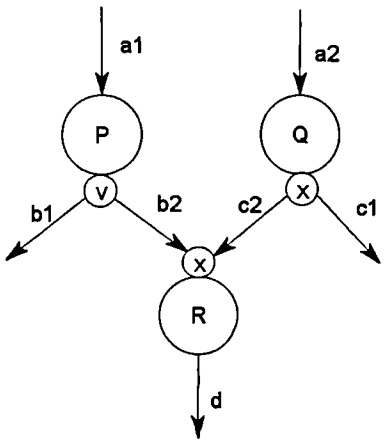
This data flow diagram is invalid becase it contains a case in which no non-internal output data flow is produced, which is not a correct way of drawing DFDs. From the diagram above, if the process P produces only the output data flow b2 and the process Q produces only the output data flow c2, the output data flow d cannot be produced (according to the diagram above, the output data flow d is produced if and only if either the input data flow b2 or c2 but not both flows into the process R). Therefore, in this case no non-internal output data flow is produced.

4)



Similar to 3), if the process P produces only the output data flow b2 and the process Q produces only the output data flow c2, the output data flow d cannot be produced.

5)



Similar to 3), if the process P produces only the output data flow b2 and the process Q produces only the output data flow c2, the output data flow d cannot be produced.

4.5.4 Inconsistency between the outputs specified in the interface specification and the outputs specified in the process specification

In each bottom level operation schema (the schema which defines the bottom level process on the RDFD), the outputs specified in the interface specification must be consistent with the detailed specifications defined in that schema. Therefore, formal proofs should be applied in order to detect this error.

For example, from the library system, the schema "R1_1_1_Check_requestor" is given as follow.

R1_1_1_Check_requestor

⇧ R1_1_Check_copy_check_out (Requestor?, Valid_requestor!, Invalid_requestor!)
Library_state (Ξ Staff)

Requestor? ➲ Valid_requestor! ⊗ Invalid_requestor!
(Requestor? ∈ Staff ∧ Valid_requestor! = True)
∨ (Requestor? ∉ Staff ∧ Invalid_requestor! = Copy_check_out_invalid_requestor)

The interface specification is

Requestor? ➲ Valid_requestor! ⊗ Invalid_requestor!

and the detailed operations are

(Requestor? ∈ Staff ∧ Valid_requestor! = True)
∨ (Requestor? ∉ Staff ∧ Invalid_requestor! = Copy_check_out_invalid_requestor)

Therefore we must prove that either "Valid_requestor!" or "Invalid_requestor" but not both of them is produced as the output. In fact, the values of the outputs are unimportant for this type of proof - it is only whether or not they are produced that matters. This means that control flows, such as "Valid_requestor!", and data flows, such as "Invalid_requestor!", can be treated uniformly.

From the specifications given above, this can be proved correct. In order to prove this, first the precondition of each output is calculated. Then from the preconditions, the proof can be shown.

The precondition of the output "Valid_requestor!" is

Requestor? ∈ Staff

The precondition of the output "Invalid_requestor!" is

Requestor? ∉ Staff

Since these two predicates cannot be both true at the same time, it confirms that only one of the two outputs is produced but not both.

However, if the detailed operations are defined as

Requestor? ∈ Staff
Valid_requestor! = True
Invalid_requestor! = Copy_check_out_invalid_requestor

In this case, the two outputs have the same precondition which is

Requestor? ∈ Staff

Therefore, both outputs are produced which contradicts with the interface specification. So, the two specifications are inconsistent.

4.5.5 Inconsistency between the process and the critical requirements

The critical requirements of a system are requirements that must hold throughout the system time and the processes defined must not effect the validation of the critical requirements. In other words, the critical requirements must always be valid regardless of all the operations performed.

However, the operation specifications defined in the operation schemas may not be consistent with the critical requirements defined. Therefore, this kind of inconsistency must be detected.

It is shown in section 4.5.1 how the critical requirements can be extracted from the state schemas. For example, from the library system, the critical requirements are:

<Borrowers, Staff> partition Users
<Checked_out_copies, Available_copies> partition Library_copies
dom writes = Library_book_authors
ran writes = Library_books
dom Is_a_subject_of = Library_book_subjects
ran Is_a_subject_of = Library_books
dom Is_a_copy_of = Library_copies
ran Is_a_copy_of = Library_books
$\forall c$: Library_copies • # Is_a_copy_of (| {c} |) = 1
dom Currently_checks_out $\subseteq$ Borrowers
ran Currently_checks_out = Checked_out_copies
$\forall c$: Checked_out_copies • # Currently_checks_out $^{-1}$ (| {c} |) = 1
Max_copy > 0
$\forall b$: dom Currently_checks_out • # Currently_checks_out (| {b} |) $\leq$ Max_copy

For every operation schema that causes some changes on the state space of the system, it must be proved that the after state of the state space is consistent with the critical requirements.

For example, from the library system the schema "R1_2_Record_copy_check_out" causes some changes on the state space of the system as defined in that schema as follows.

Available_copies ' = Available_copies \ {Copy?}
Checked_out_copies ' = Checked_out_copies $\cup$ {Copy?}
Currently_checks_out ' = Currently_checks_out $\cup$ {Borrower? $\mapsto$ Copy?}

Therefore, the proofs are required to show that these operations preserve the critical requirements.

The operations above may effect the following critical requirements:

<Checked_out_copies, Available_copies> partition Library_copies
dom Currently_checks_out $\subseteq$ Borrowers
ran Currently_checks_out = Checked_out_copies
$\forall c$: Checked_out_copies • # Currently_checks_out $^{-1}$ (| {c} |) = 1
$\forall b$: dom Currently_checks_out • # Currently_checks_out (| {b} |) $\leq$ Max_copy

Therefore, formal proofs are required to show that the operations do not cause any inconsistency in the critical requirements.

For example, we must prove that

<Checked_out_copies', Available_copies'> partition Library_copies'

or in fact we must prove that

Checked_out_copies' ∩ Available_copies' = ∅

The proofs can be shown as follows.

1. Checked_out_copies' ∩ Available_copies'
2. (Available_copies \ {Copy?}) ∩ (Checked_out_copies ∪ {Copy?})
3. Available_copies ∩ Checked_out_copies
4. ∅

Chapter 5 Software design specification technique

5.1 Overview

This chapter presents a technique for specifying software design. The proposed software design specification technique uses the same specification languages as the proposed software requirements specification technique described in chapter 3.

Section 5.2 gives an overview of the proposed software design specification technique. First, it discusses two sorts of design decisions usually taken when a software design specification is developed from a software requirements specification: data refinement (data design) and process refinement (process design). Then, the steps of the proposed SDS technique are given. Section 5.3 describes how to draw DERDs and DDSDs to define the static aspects of the system as designed by the designers. Section 5.4 describes how to draw DDFDs and DDSDs to define the dynamic aspects of the system as designed by the designers. Finally, section 5.5 describes how to write DZs to define formally both the static and dynamic aspects of the system designed.

5.2 Overview of the proposed SDS technique

5.2.1 From requirements to designs

In an SRS, a software system is specified by describing what the software will do to achieve the end-users' requirements without describing any design decision. The main objective of an SRS is to specify a complete definition of the necessary capabilities of the system from the end-users' point of view. It specifies the data and processes to be supported by the system from the end-users' perspective. In an SRS, the data are specified using abstract data types and processes are specified with an aim to convey end-users' requirements not designs. The abstract data types are used because of their expressive clarity rather than their ability to be directly represented in a computer. At this stage, we shall not use concrete data types (data types which can be directly represented in a computer); or in other words we shall not consider how the data will be efficiently represented in a computer, this will be dealt with later on in the design stage. Similarly, the required processes are specified with an aim to convey the required operations of a system; what sort of processes will result in the most efficient system is not yet brought into our attention at this stage.

In an SDS, the software system is specified by describing how the software will do in order to satisfy the requirements stated in the SRS. The main objective of an SDS is to record the result of the design processes (data refinement and process refinement) or in other words to record the transformation of the abstract data structures and required processes specified in the SRS into a set of concrete data structures and designed processes.

To develop an SDS from an SRS, two sorts of design decisions are usually taken: *data refinement* (*data design*) and *process refinement* (*process design*) [2, 48, 86]. For a simple system, it is possible that the refinement of an SRS to a final SDS can be done in one step,

which is called *direct refinement* [86]. For a complex system, it is likely that the refinement of an SRS to a final SDS must be done in more than one step, which is called *deferred refinement* [86], to allow the design decisions to be recorded, clearly, step by step.

5.2.2 Steps of the proposed SDS technique

The proposed software design specification technique, which will be described in this chapter, is similar to the proposed software requirements specification technique described in chapter 3. The same specification languages are used in both an SRS and SDS. Therefore, the steps for producing an SDS is quite similar to those of an SRS.

It is more suitable, if it is possible, to use the same specification languages to describe both an SRS and SDS. This is because an SDS can be viewed as a refinement of an SRS and by using the same specification languages it is more convenient in producing an SDS and also it is easier to reason about the relationships between the SRS and the corresponding SDS. However, to be able to use the same specification languages, the specification languages being used must have the ability to record both requirements and designs; otherwise it would not be possible.

As mentioned earlier the proposed SDS technique uses the same four specification languages as the proposed SRS technique. The four specification languages are: ERDs, DFDs, DSDs, and Z. In both an SRS and SDS, a system is specified in terms of the static and dynamic aspects of that system. As already shown in chapter 3, the static aspects of a system are specified graphically by using ERDs and DSDs, and formally by using Z specifications; the dynamic aspects of a system are specified graphically by using DFDs and DSDs, and formally by using Z specifications. As mentioned in section 2.6, we call the ERDs, DFDs, DSDs, and Z specifications which are produced to specify software designs the DERDs, DDFDs, DDSDs, and DZs accordingly.

The steps of the proposed SRS and SDS technique are very similar. The proposed SDS technique is applied using the following steps:

1) Draw DERDs and DDSDs
   - 1.1) Identify all entities and their relationships
   - 1.2) Draw DERDs
   - 1.3) Draw DDSDs

2) Draw DDFDs and DDSDs
   - 2.1) Draw the context DDFD and DDSDs
     - 2.1.1) Identify all external entities and input and output data flows
     - 2.1.2) Draw the context DDFD
     - 2.1.3) Identify the data components of each input and output data flow
     - 2.1.4) Draw DDSDs
   - 2.2) Draw the next level DDFDs and DDSDs
     - 2.2.1) Identify sub-processes
     - 2.2.2) Identify input and output data flows of each sub-process
     - 2.3.3) Draw the next level DDFDs
     - 2.2.4) Identify the data components of each new internal data flow
     - 2.3.5) Draw DDSDs

3) Write DZs
    3.1) Define the state of the system
    3.2) Define the initial state of the system
    3.3) Define the operations of the system

In the following sections, each step given above will be demonstrated, and the SRS of the library system as produced in chapter 3 will be used to show how an SDS can be developed from the SRS. To keep the example system simple, only the data refinement (data design) process is done to refine the SRS into the SDS; the processes of the required system are not directly refined. However, in real practice the processes of the required system may also be refined (designed).

5.3 Step 1: draw DERDs and DDSDs

5.3.1 Step 1.1: identify all entities and their relationships

During the data design process, the concrete data types are selected to implement the abstract data types in the SRS. For example, the concrete data type "array" may be selected to implement the abstract data type "set". In general, it is possible that an abstract data type can be implemented by various concrete data types. It is the task of the designers to analyze and, then, make the decision. Such a task is beyond the scope of this thesis.

Relational data analysis or *normalization* is one of the most widely used data design techniques and several modern system development methodologies use this technique [2]. The proposed SDS technique is developed specifically to support this technique.

Relational data analysis or normalization is a formal method of converting data structures into a set of well-designed relations [63]. The relations derived from applying the relational data analysis technique will be depicted on DERDs.

Concerning the library system, let us assume that the designers have made the design decisions about the entities and relationships of the system as follows:

1) The abstract entity "Users" in the SRS will not be implemented, however the abstract entities "Borrowers" and "Staff" will be implemented by two concrete entities (relations/databases) "D_borrowers" and "D_staff". The data type of the concrete entity "D_borrowers" is "ℙ D_BORROWER". The data type "D_BORROWER" has 3 attributes: "D_id" whose type is "D_PERSON_ID; "D_name" whose type is "D_NAME"; "D_reg_dt" whose type is "D_DATE". The primary key of this concrete entity is "D_id". The data type of the concrete entity "D_staff" is "ℙ D_STAFF". The data type "D_STAFF" has 3 attributes: "D_id" whose type is "D_PERSON_ID"; "D_name" whose type is "D_NAME"; "D_salary" whose type is "ℕ". The primary key of this concrete entity is "D_id".

2) The abstract entity "Library_book_authors" will be implemented by the concrete entity "D_library_book_authors". The data type of the concrete entity "D_library_book_authors" is "ℙ D_AUTHOR". The data type "D_AUTHOR" has 2 attributes: "D_id" whose type is "D_AUTHOR_ID"; "D_name" whose type is "D_AUTHOR_NAME". The primary key of this concrete entity is "D_id".

3) The abstract entity "Library_book_subjects" will be implemented by the concrete entity "D_library_book_subjects". The data type of the concrete entity "D_library_book_subjects" is "ℙ D_SUBJECT". The data type "D_SUBJECT" has 2 attributes: "D_id" whose type is "D_SUBJECT_ID"; "D_name" whose type is "D_SUBJECT_NAME". The primary key of this concrete entity is "D_id".

4) The abstract entity "Library_books" will be implemented by the concrete entity "D_library_books". The data type of the concrete entity "D_library_books" is "ℙ D_BOOK". The data type "D_BOOK" has 4 attributes: "D_id" whose type is "D_BOOK_ID"; "D_isbn" whose type is "D_ISBN"; "D_issn" whose type is "D_ISSN"; "D_title" whose type is "D_TITLE". The attributes "D_isbn" and "D_issn" are in exclusive disjunction which means that only one of them will exist but not both. The primary key of this concrete entity is "D_id".

5) The abstract entities "Checked_out_copies" and "Available_copies" will not be directly implemented, but the abstract entity "Library_copies" will be implemented by the concrete entity "D_library_copies". The data type of the concrete entity "D_library_copies" is "ℙ D_COPY". The data type "D_COPY" has 2 attributes: "D_id" whose type is "D_COPY_ID"; "D_status" whose type is "D_STATUS". The primary key of this concrete entity is "D_id".

6) The abstract relationship "Writes" will be implemented by the concrete relationship (will be implemented as a relation) "D_writes". The concrete relationship "D_writes" has 2 attributes: the first attribute whose type is "D_AUTHOR_ID"; the second attribute whose type is "D_BOOK_ID".

7) The abstract relationship "Is_a_subject_of" will be implemented by the concrete relationship "D_is_a_subject_of". The concrete relationship "D_is_a_subject_of" has 2 attributes: the first attribute whose type is "D_SUBJECT_ID"; the second attribute whose type is "D_BOOK_ID".

8) The abstract relationship "Is_a_copy_of" will not be implemented by a relation, but it will be implemented by including primary keys of the relation "D_library_books" in the concrete entity "D_library_copies".

9) The abstract relationship "Currently_checks_out" will be implemented by the concrete relationship "D_currently_checks_out". The concrete relationship "D_currently_checks_out" has 2 attributes: the first attribute whose type is "D_PERSON_ID"; the second attribute whose type is "D_COPY_ID".

5.3.2 Step 1.2: draw DERDs

Next, DERDs are drawn to capture the entities and relationships as designed in the previous step. The DERDs are drawn by following the notations given in section 2.2.1.

The DERD of the library system is drawn as shown in Figure 5-1.

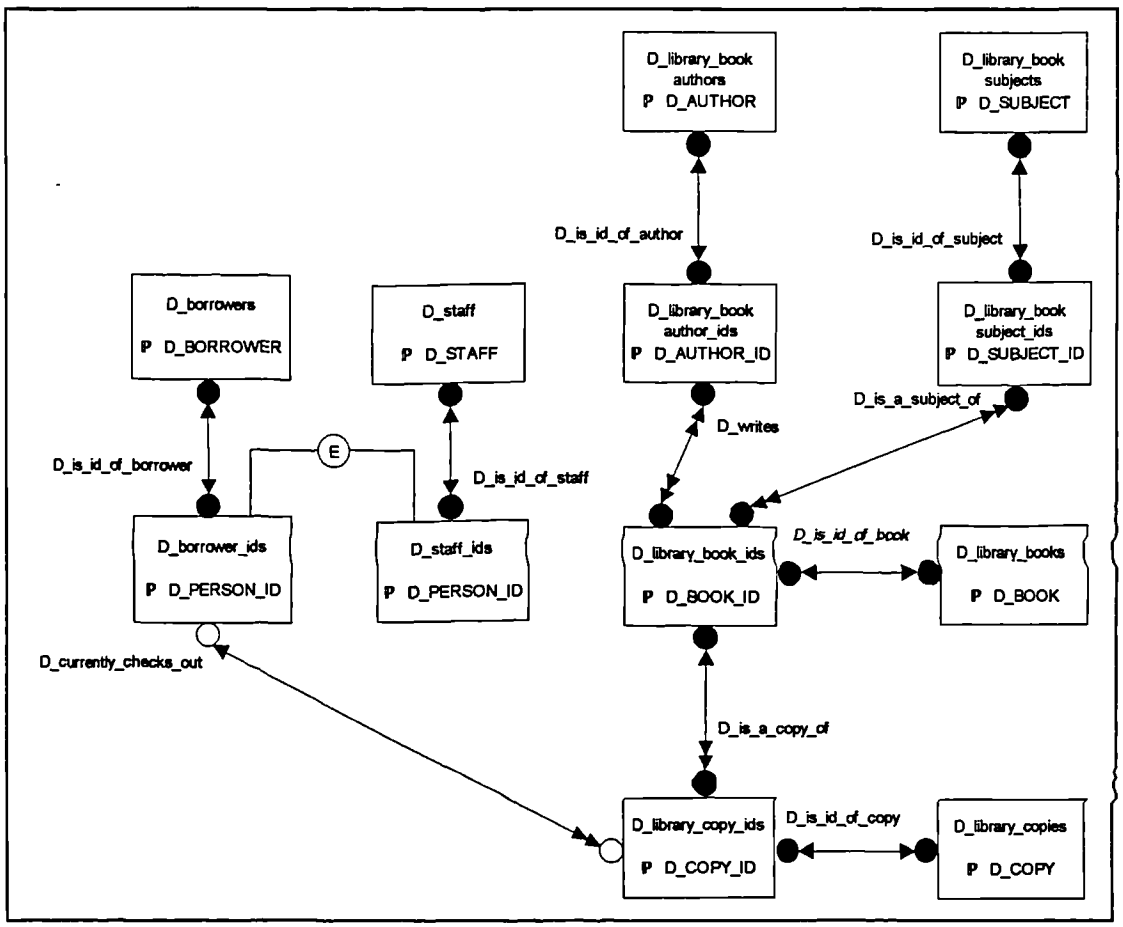


Figure 5-1: The DERD of the library system

5.3.3 Step 1.3: draw DDSDs

Then a DDSD is drawn to show the attributes of each data type as designed in step 1.1.

The DDSDs of the data types as designed in step 1.1 are shown in Figures 5-2 to 5-7.

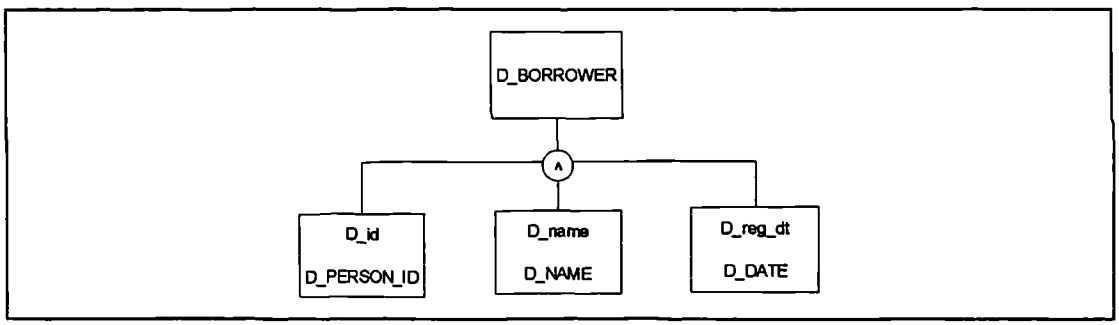


Figure 5-2: The DDSD of the data type "D_BORROWER"

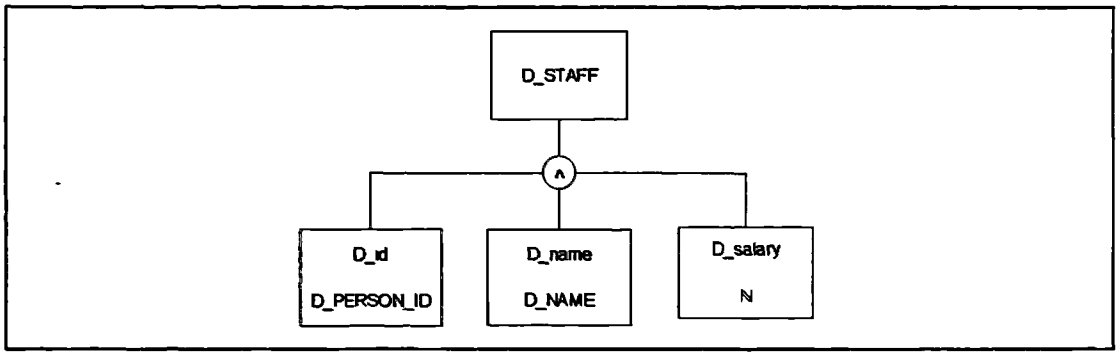


Figure 5-3: The DDSD of the data type "D_STAFF"

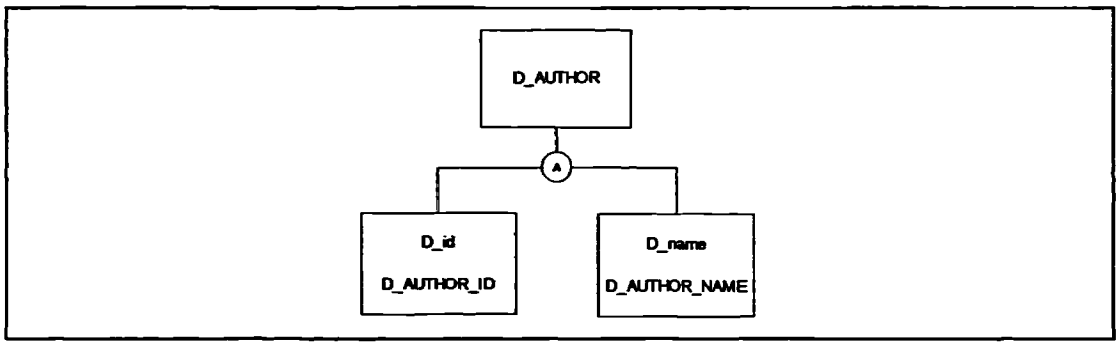


Figure 5-4: The DDSD of the data type "D_AUTHOR"

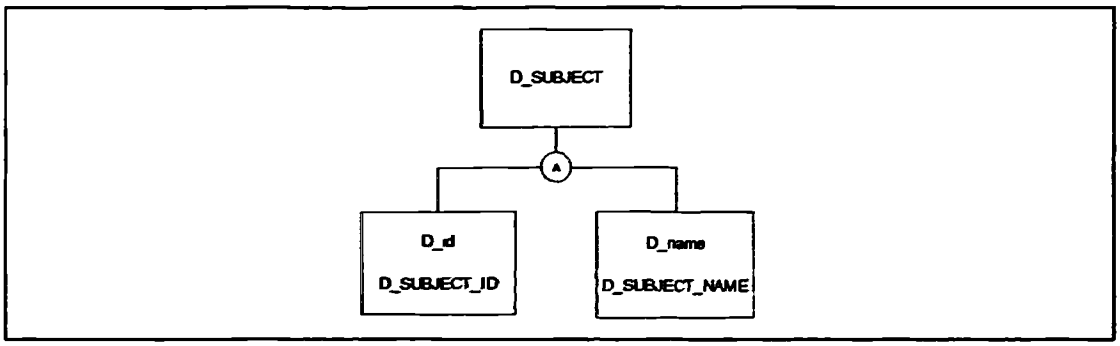


Figure 5-5: The DDSD of the data type "D_SUBJECT"

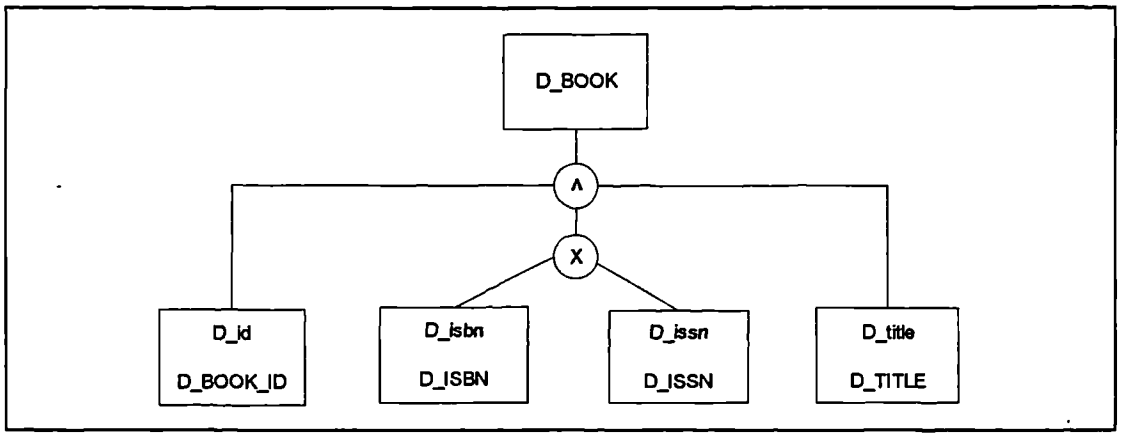


Figure 5-6: The DDSD of the data type "D_BOOK"

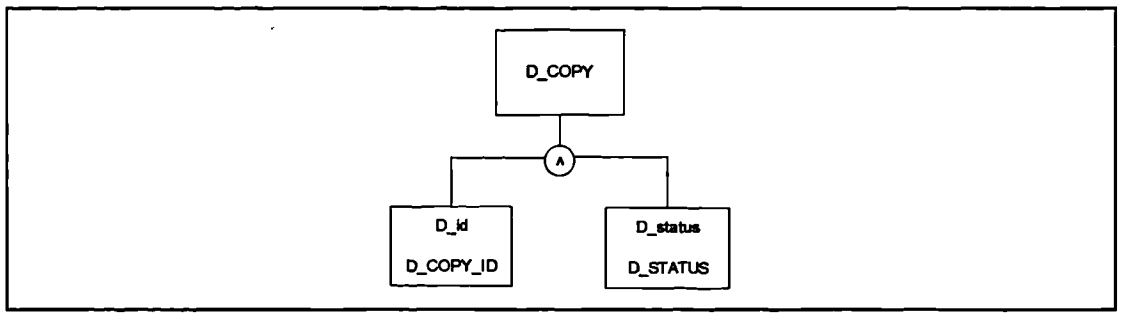


Figure 5-7: The DDSD of the data type "D_BORROWER"

5.4 Step 2: draw DDFDs and DDSDs

5.4.1 Step 2.1: draw the context DDFD and DDSDs

5.4.1.1 Step 2.1.1: identify all external entities and input and output data flows

In this step, some input and output data flows as shown on the RDFDs may be implemented by the concrete data flows. However, some input and output data flows on the RDFDs may remain the same since the designers have decided that it is not necessary to refine them.

Concerning the library system, for example, the input data flow "Copy_check_out" on the RDFDs will be implemented by the concrete data flow "D_copy_check_out". However, the output data flow "Valid_copy_check_out" on the RDFDs is not refined, therefore the same name is used.

5.4.1.2 Step 2.1.2: draw the context DDFD

Then, the context DDFD of the system is drawn.

The context DDFD of the library system is shown in Figure 5-8.

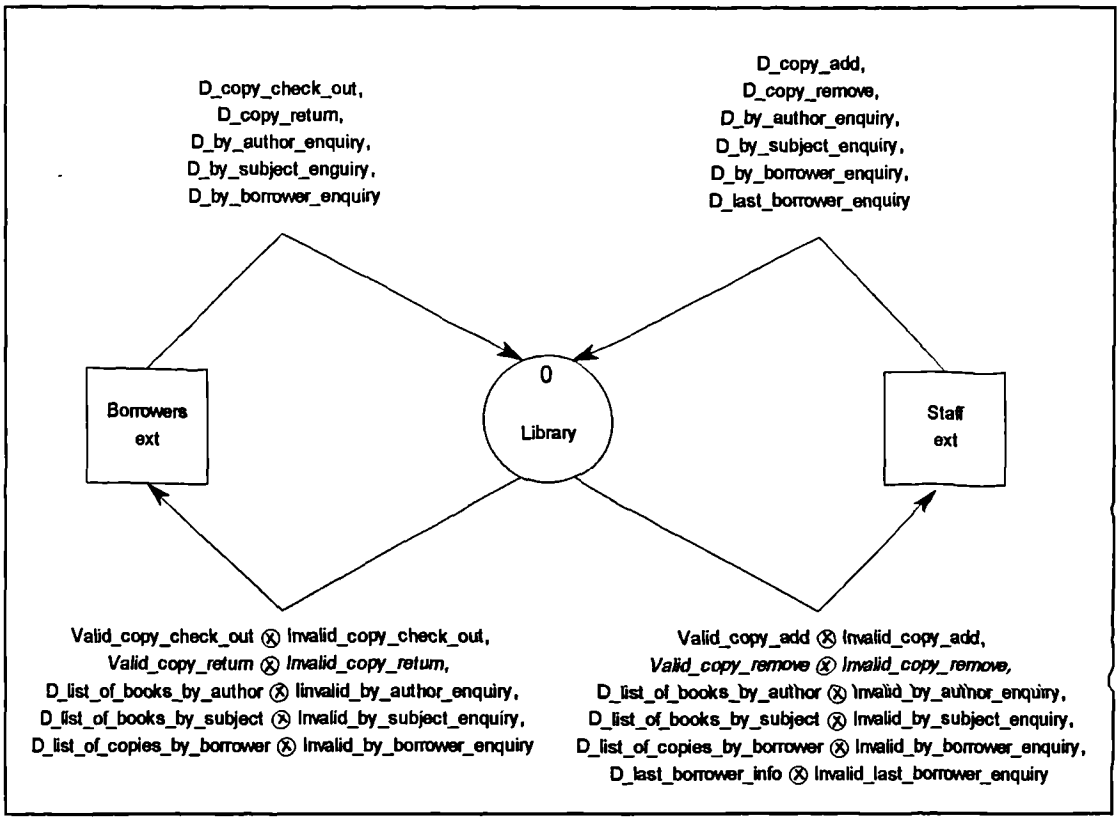


Figure 5-8: The context DDFD of the library system

5.4.1.3 Step 2.1.3: identify the data components of each input and output data flow

Next, the data components of each input and output data flow must be identified as well as the interfaces of those data components. Then each data component is assigned a data type.

Concerning the library system, for example, the input data flow "D_copy_check_out" is defined as having three attributes: "D_requestor_id", "D_borrower_id", and "D_copy_id". Then the data type is assigned to each attribute.

The data components of other input and output data flows can be identified in a similar way.

5.4.1.4 Step 2.1.4: draw DDSDs

Next, a DDSD is drawn to depict the data structure of each data flow.

The DDSD of the input data flow "D_copy_check_out" is shown in Figure 5-9. The DDSDs of other input and output data flows can be drawn in a similar way.

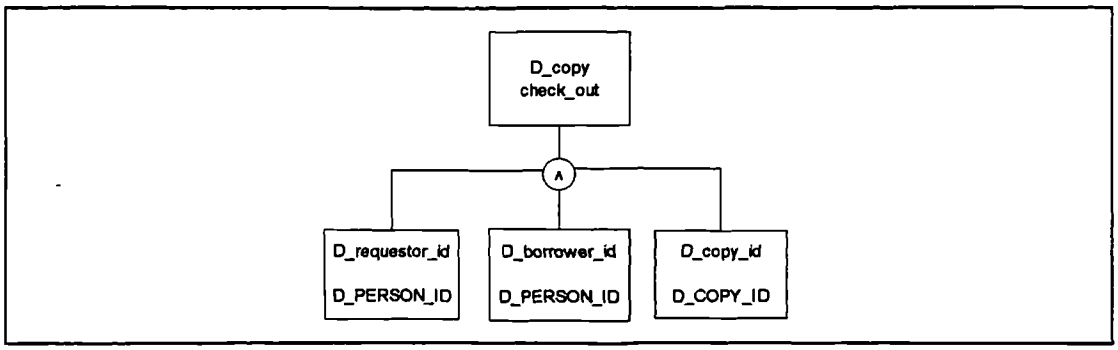


Figure 5-9: The DDSD of the data flow "D_copy_check_out"

5.4.2 Step 2.2: draw the next level DDFDs and DDSDs

Steps 2.2.1 to 2.2.5 are repeated for each next level DDFD and DDSDs.

5.4.2.1 Step 2.2.1: identify sub-processes

Since only data refinement is considered, sub-processes of the DDFDs are the same as the RDFDs.

5.4.2.2 Step 2.2.2: identify input and output data flows of each sub-process

The input and output data flows of each sub-process on the DDFDs are the same as the RDFDs, except that the abstract data flows are replaced by the concrete data flows and the abstract entities are replaced by the concrete entities.

5.4.2.3 Step 2.2.3: draw the next level DDFD

Then, the next level DDFD is drawn.

Fig 5-10 shows the decomposition of the process 0 "Library" into five sub-processes.

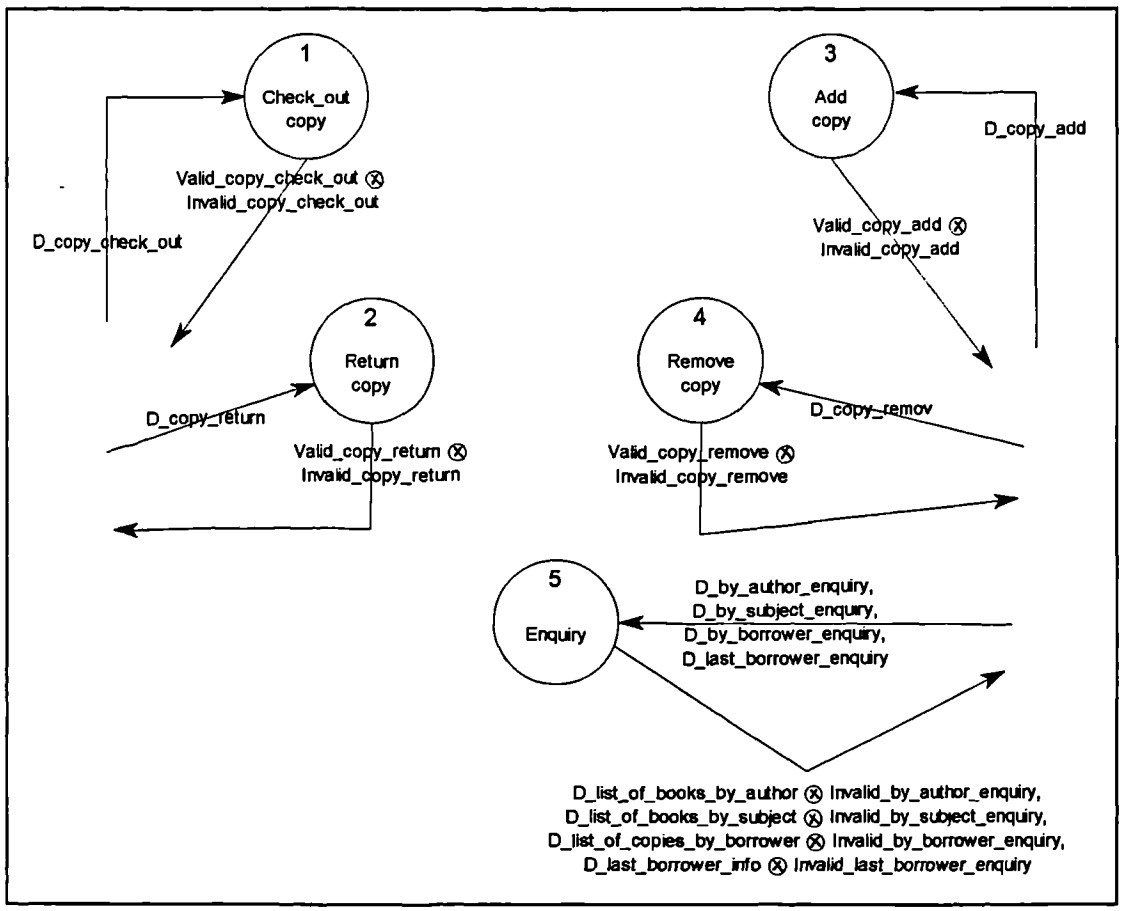


Figure 5-10: The decomposition of the process 0 DDFD

5.4.2.4 Step 2.2.4: identify the data components of each new internal data flow

The data components of each new internal data flow are identified.

5.4.2.5 Step 2.2.5: draw DDSDs

Then, DDSDs of the new internal data flows are drawn.

Steps 2.2.1 to 2.2.5 are repeated until the details desired are obtained.

The DDFDs and DDSDs of the process 1 "Check_out_copy" is shown as follows. The DDFDs and DDSDs of other processes can be derived in a similar way.

A. Check out a copy of a book

Figure 5-11 shows the decomposition of the process 1 "Check_out_copy".

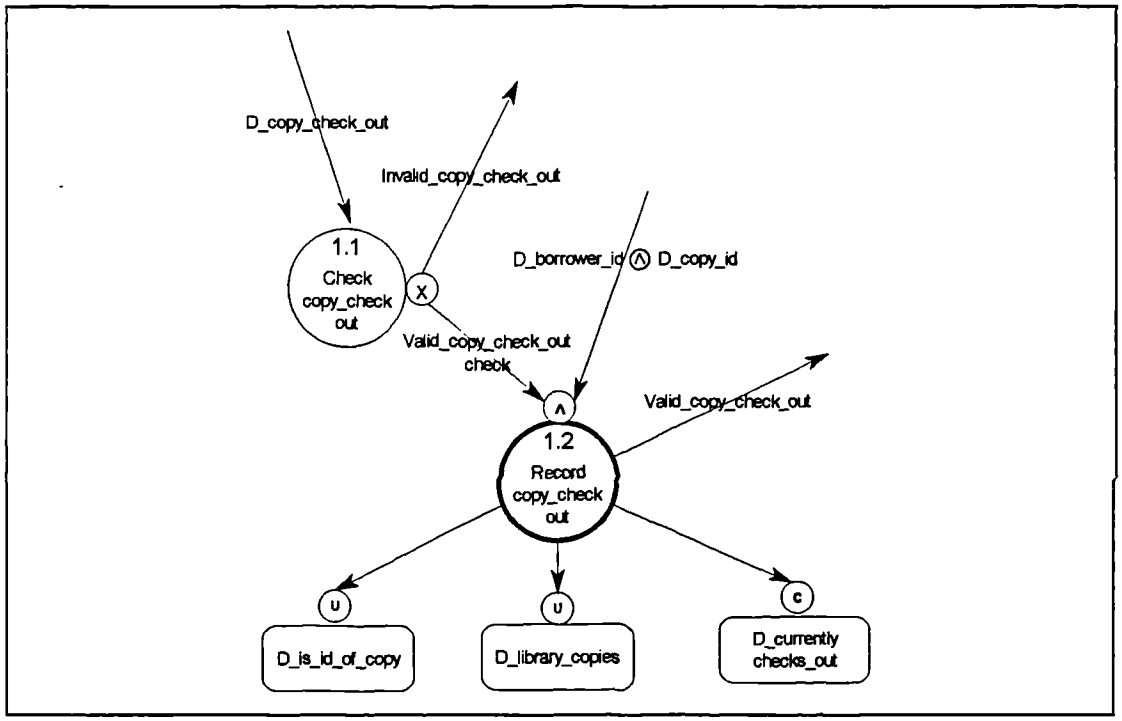


Figure 5-11 The decomposition of the process 1 DDFD

Figure 5-12 shows the decomposition of the process 1.1 DDFD.

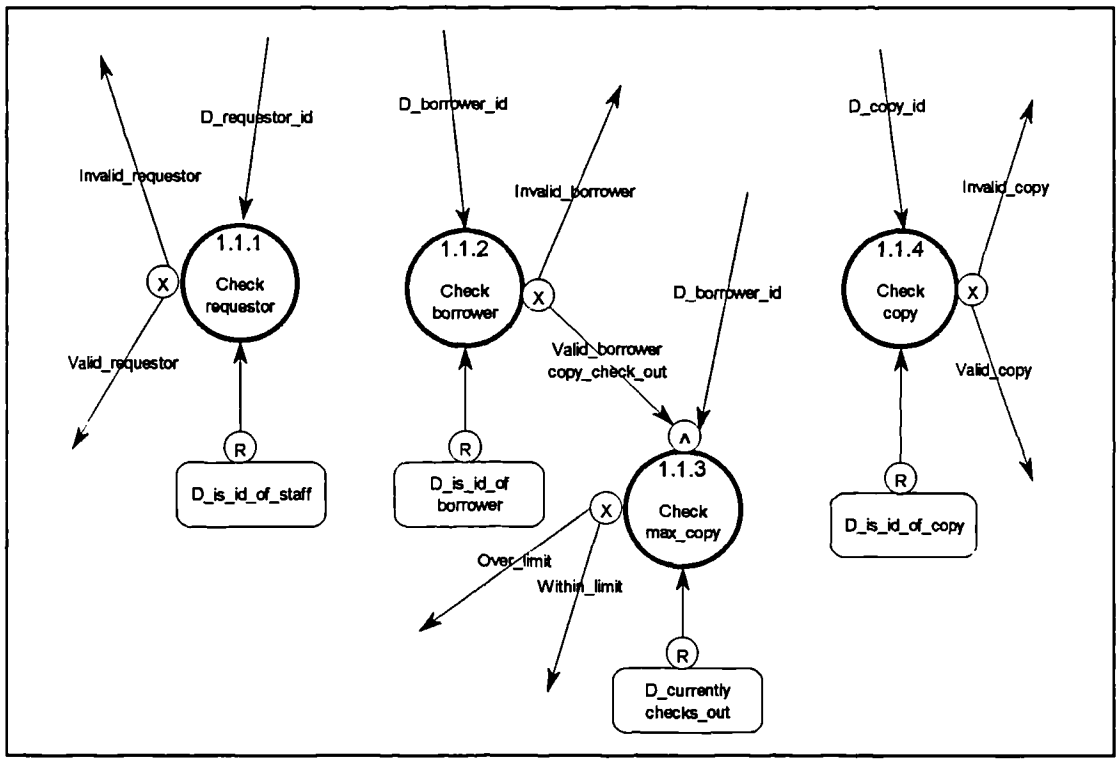


Figure 5-12: The decomposition of process 1.1 DDFD

5.5 Step 3: write DZs

5.5.1 Step 3.1: define the state of the system

From the DERDs and DDSDs drawn in the previous steps, if the data type has data components (as shown on the DDSD), that data type will be defined as a schema type in Z; otherwise it may be defined either as a basic type or free type.

Concerning the library system, the following basic data types are defined.

[D_PERSON_ID, D_NAME, D_DATE, D_AUTHOR_ID, D_AUTHOR_NAME, D_SUBJECT_ID, D_SUBJECT_NAME, D_BOOK_ID, D_ISBN, D_ISSN, D_TITLE, D_COPY_ID]

There is one free type defined as follow.

D_STATUS ::= Available | Checked_out

There are six schema types defined as follows.

D_BORROWER

D_id : D_PERSON_ID ⓐ
D_name : D_NAME ⓐ
D_reg_dt : D_DATE

D_STAFF

D_id : D_PERSON_ID ⓐ
D_name : D_NAME ⓐ
D_salary : ℕ

D_AUTHOR

D_id : D_AUTHOR_ID ⓐ
D_name : D_AUTHOR_NAME

D_SUBJECT

D_id : D_SUBJECT_ID ⓐ
D_name : D_SUBJECT_NAME

D_BOOK

D_id : D_BOOK_ID ⓐ
(D_isbn : D_ISBN ⓧ
D_issn : D_ISSN) ⓐ
D_title : D_TITLE

D_COPY

D_id : D_COPY_ID ⓐ
D_status : D_STATUS

Then, the state of the system is defined.

The state of the library system can be defined in Z as follows.

D_person_state

D_borrowers : ℙ D_BORROWER
D_borrower_ids : ℙ D_PERSON_ID
D_staff : ℙ D_STAFF
D_staff_ids : ℙ D_PERSON_ID
D_is_id_of_borrower : D_borrower_ids •↔• D_borrowers
D_is_id_of_staff : D_staff_ids •↔• D_staff

∀ Borrower_id : D_borrower_ids •
Borrower_id = (D_is_id_of_borrower (Borrower_id)).D_id
∀ Staff_id : D_staff_ids •
Staff_id = (D_is_id_of_staff (Staff_id)).D_id
disjoint ⟨D_borrower_ids, D_staff_ids⟩

The concrete entities "D_borrowers" and "D_staff" are defined. The relationship "D_is_id_of_borrower" allows each instance of the entity "D_borrowers" to be retrieved by the attribute "D_id". Similarly, the relationship "D_is_id_of_staff" allows each instance of the entity "D_borrowers" to be retrieved by the attribute "D_id".

D_author_state

D_library_book_authors : ℙ D_AUTHOR
D_library_book_author_ids : ℙ D_AUTHOR_ID
D_is_id_of_author : D_library_book_author_ids •↔• D_library_book_authors

---

∀ Author_id : D_library_book_author_ids •
Author_id = (D_is_id_of_author (Author_id)).D_id

D_subject_state

D_library_book_subjects : ℙ D_SUBJECT
D_library_book_subject_ids : ℙ D_SUBJECT_ID
D_is_id_of_subject : D_library_book_subject_ids •↔• D_library_book_subjects

---

∀ Subject_id : D_library_book_subject_ids •
Subject_id = (D_is_id_of_subject (Subject_id)).D_id

D_book_state

D_library_books : ℙ D_BOOK
D_library_book_ids : ℙ D_BOOK_ID
D_is_id_of_book : D_library_book_ids •↔• D_library_books

∀ Book_id : D_library_book_ids •
Book_id = (D_is_id_of_book (Book_id)).D_id

D_copy_state

$D\_library\_copies : \mathbb{P}\ D\_COPY$
$D\_library\_copy\_ids : \mathbb{P}\ D\_COPY\_ID$
$D\_is\_id\_of\_copy : D\_library\_copy\_ids \bullet\!\!\leftrightarrow\!\!\bullet\ D\_library\_copies$

$\forall\ Copy\_id : D\_library\_copy\_ids \bullet$
$\quad Copy\_id = (D\_is\_id\_of\_copy\ (Copy\_id)).D\_id$

Then we write a schema to define the total state of the system.

D_library_state

D_person_state
D_author_state
D_subject_state
D_book_state
D_copy_state
$D\_writes : D\_library\_book\_author\_ids \bullet\!\!\leftrightarrow\!\!\gg\!\!\bullet\ D\_library\_book\_ids$
$D\_is\_a\_subject\_of : D\_library\_book\_subject\_ids \bullet\!\!\leftrightarrow\!\!\gg\!\!\bullet\ D\_library\_book\_ids$
$D\_is\_a\_copy\_of : D\_library\_copy\_ids \bullet\!\!\leftrightarrow\!\!\gg\!\!\bullet\ D\_library\_book\_ids$
$D\_currently\_checks\_out : D\_borrower\_ids\ o\!\!\leftrightarrow\!\!\gg\!\!o\ D\_library\_copy\_ids$
$Max\_copy : \mathbb{N}_1$

$\forall Copy\_id : ran\ D\_currently\_checks\_out \bullet$
$\quad (D\_is\_id\_of\_copy\ (Copy\_id)).D\_status = Checked\_out$
$\forall Borrower\_id : dom\ D\_currently\_checks\_out \bullet$
$\quad \#\ D\_currently\_checks\_out\ (|\ \{Borrower\_id\}\ |) \leq Max\_copy$

5.5.2 Step 3.2: define the initial state of the system

The initial state of the system can be defined as follows.

D_init_library_state

D_library_state

$D\_borrowers = \varnothing$
$D\_borrower\_ids = \varnothing$
$D\_staff = \varnothing$
$D\_staff\_ids = \varnothing$
$D\_library\_book\_authors = \varnothing$
$D\_library\_book\_author\_ids = \varnothing$
$D\_library\_book\_subjects = \varnothing$
$D\_library\_book\_subject\_ids = \varnothing$
$D\_library\_books = \varnothing$
$D\_library\_book\_ids = \varnothing$
$D\_library\_copies = \varnothing$
$D\_library\_copy\_ids = \varnothing$
$D\_is\_id\_of\_borrower = \varnothing$
$D\_is\_id\_of\_staff = \varnothing$
$D\_is\_id\_of\_author = \varnothing$

D_is_id_of_subject = ∅
D_is_id_of_book = ∅
D_is_id_of_copy = ∅
D_writes = ∅
D_is_a_subject_of = ∅
D_is_a_copy_of = ∅
D_currently_checks_out = ∅
Max_copy = 10

5.5.3 Step 3.3: define the operations of the system

Then, each process in the DDFDs will be translated into a schema in Z.

There is no refinement for the data type "MSG".

Then, the process 0 DDFD is translated into a schema as follows.

D0_Library

D_copy_check_out? : (D_requestor_id : D_PERSON_ID ⊗
D_borrower_id : D_PERSON_ID ⊗ D_copy_id : D_COPY_ID)
D_copy_return? : (D_requestor_id : D_PERSON_ID ⊗ D_copy_id : D_COPY_ID)
D_copy_add? : (D_requestor_id : D_PERSON_ID ⊗ D_copy_id : D_COPY_ID ⊗
D_Book : D_BOOK ⊗ [D_authors : ℙ D_AUTHOR] ⊗
[D_subjects : ℙ D_SUBJECT])
D_copy_remove? : (D_requestor_id : D_PERSON_ID ⊗
D_copy_id : D_COPY_ID)
D_by_author_enquiry? : (D_requestor_id : D_PERSON_ID ⊗
D_auhtor_id : D_AUTHOR_ID)
D_by_subject_enquiry? : (D_requestor_id : D_PERSON_ID ⊗
D_subject_id : D_SUBJECT_ID)
D_by_borrower_enquiry? : (D_requestor_id : D_PERSON_ID ⊗
D_borrower_id : D_PERSON_ID)
D_last_borrower_enquiry? : (D_requestor_id : D_PERSON_ID ⊗
D_copy_id : D_COPY_ID)
Valid_copy_check_out! : MSG
Invalid_copy_check_out! : (Invalid_requestor : MSG ⊕
(Invalid_borrower : MSG ⊗ Over_limit : MSG) ⊕ Invalid_copy : MSG)
Valid_copy_return! : MSG
Invalid_copy_return! : (Invalid_requestor : MSG ⊕ Invalid_copy : MSG)
Valid_copy_add! : MSG
Invalid_copy_add! : (Invalid_requestor : MSG ⊕ Invalid_copy : MSG)
Valid_copy_remove! : MSG
Invalid_copy_remove! : (Invalid_requestor : MSG ⊕ Invalid_copy : MSG)
D_list_of_books_by_author! : ℙ D_BOOK
Invalid_by_author_enquiry! : (Invalid_requestor : MSG ⊕ Invalid_author : MSG)
D_list_of_books_by_subject! : ℙ D_BOOK
Invalid_by_subject_enquiry! : (Invalid_requestor : MSG ⊕ Invalid_subject : MSG)
D_list_of_copies_by_borrower! : ℙ D_COPY_ID
Invalid_by_borrower_enquiry! : ((Invalid_user : MSG ⊗
Unauthorized_requestor : MSG) ⊕ Invalid_borrower : MSG)
D_last_borrower_info! : D_PERSON
Invalid_last_borrower_enquiry! : (Invalid_requestor : MSG ⊕
Invalid_copy : MSG)
⇐ Borrowers_ext (D_copy_check_out?, D_copy_return?,
D_by_author_enquiry?, D_by_subject_enquiry?,
D_by_borrower_enquiry?,

Valid_copy_check_out!, Invalid_copy_check_out!,
Valid_copy_return!, Invalid_copy_return!,
D_list_of_books_by_author!, Invalid_by_author_enquiry!,
D_list_of_books_by_subject!, Invalid_by_subject_enquiry!,
D_list_of_copies_by_borrower!, Invalid_by_borrower_enquiry!)

⇐ Staff_ext (D_copy_add?, D_copy_remove?,
D_by_author_enquiry?, D_by_subject_enquiry?,
D_by_borrower_enquiry?, D_last_borrower_enquiry?,
Valid_copy_add!, Invalid_copy_add!,
Valid_copy_remove!, Invalid_copy_remove!,
D_list_of_books_by_author!, Invalid_by_author_enquiry!,
D_list_of_books_by_subject!, Invalid_by_subject_enquiry!,
D_list_of_copies_by_borrower!, Invalid_by_borrower_enquiry!,
D_last_borrower_info!, Invalid_last_borrower_enquiry!)

D_copy_check_out?, D_copy_return?, D_copy_add?, D_copy_remove?,
D_by_author_enquiry?, D_by_subject_enquiry?, D_by_borrower_enquiry?,
D_last_borrower_enquiry?
➲
Valid_copy_check_out!⊗Invalid_copy_check_out!,
Valid_copy_return! ⊗ Invalid_copy_return!,
Valid_copy_add! ⊗ Invalid_copy_add!,
Valid_copy_remove! ⊗ Invalid_copy_remove!,
D_list_of books_by_author! ⊗ Invalid_by_author_enquiry!,
D_list_of_books_by_subject! ⊗ Invalid_by_subject_enquiry!,
D_list_of_copis_by_borrower!⊗ Invalid_by_borrower_enquiry!,
D_last_borrower_info! ⊗ Invalid_last_borrower_enquiry!

Next, the processes 1 to 5 along with the related DDSDs can be translated into Z specifications as follows.

D1_Check_out_copy

⇧ D0_Library (D_copy_check_out?, Valid_copy_check_out!,
Invalid_copy_check_out!)

D_copy_check_out? ➲ Valid_copy_check_out! ⊗ Invalid_copy_check_out!

D2_Return_copy

⇧ D0_Library (D_copy_return?, Valid_copy_return!, Invalid_copy_return!)

D_copy_return? ➲ Valid_copy_return! ⊗ Invalid_copy_return!

D3_Add_copy

⇧ D0_Library (D_copy_add?, Valid_copy_add!, Invalid_copy_add!)

D_copy_add? ➲ Valid_copy_add! ⊗ Invalid_copy_add!

D4_Remove_copy

⇧ D0_Library (D_copy_remove?, Valid_copy_remove!, Invalid_copy_remove!)

D_copy_remove? ➲ Valid_copy_remove! ⊗ Invalid_copy_remove!

D5_Enquiry

⇧ D0_Library (D_by_author_enquiry?, D_by_subject_enquiry?,
D_by_borrower_enquiry?, D_last_borrower_enquiry?,
D_list_of_books_by_author!, Invalid_by_author_enquiry!,
D_list_of_books_by_subject!, Invalid_by_subject_enquiry!,
D_list_of_copies_by_borrower!, Invalid_by_borrower_enquiry!,
D_last_borrower_info!, Invalid_last_borrower_enquiry!)

D_by_author_enquiry?, D_by_subject_enquiry?, D_by_borrower_enquiry,
D_last_borrower_enquiry? ➲
D_list_of_books_by_author! ⊗ Invalid_by_author_enquiry!,
D_list_of_books_by_subject! ⊗ Invalid_by_subject_enquiry!,
D_list_of_copies_by_borrower! ⊗ Invalid_by_borrower_enquiry!,
D_last_borrower_info! ⊗ Invalid_last_borrower_enquiry!

Next, the processes 1.1 and 1.2 and all sub-processes of the process 1.1 can be translated into Z specifications as follows.

D1_1_Check_copy_check_out

⇧ D1_Check_out_copy (D_copy_check_out?, Invalid_copy_check_out!)
Valid_copy_check_out_check! : (Valid_requestor : 𝔹 ⊙ Within_limit : 𝔹 ⊙
Valid_copy : 𝔹)

D_copy_check_out? ➲ Valid_copy_check_out_check! ⊗
Invalid__copy_check_out!

D1_2_Record_copy_check_out

⇧ D1_Check_out_copy (D_borrower_id?, D_copy_id?, Valid_copy_check_out!)
⇨ D1_1_Check_copy_check_out (Valid_copy_check_out_check?)
Library_state (Δ D_is_id_of_copy, Δ D_library_copies, Δ D_currently_checks_out)

D_borrower_id? ⊙ D_copy_id? ⊙ Valid_copy_check_out_check? ➲
Valid_copy_check_out!
D_is_id_of_copy ' = D_is_id_of_copy ⊕
{D_copy_id? ↦ < D_copy_id?, Checked_out >}
D_library_copies ' = D_library_copies \ {D_is_a_copy_of (D_copy_id?)} ∪
{<D_copy_id?, Checked_out>}
D_currently _checks_out ' = D_currently_checks_out ∪
{D_borrower_id? ↦ D_copy_id?}
Valid_copy_check_out! = Copy_check_out_ok

D1_1_1_Check_requestor

⇧ D1_1_Check_copy_check_out (D_requestor_id?, Valid_requestor!, Invalid_requestor!)
D_library_state (Ξ D_is_id_of_staff)

D_requestor_id? ➲ Valid_requestor! ⊗ Invalid_requestor!
(D_requestor_id? ∈ dom D_Is_id_of_staff ∧ Valid_requestor! = True)
∨ (D_requestor_id? ∉ dom D_is_id_of_staff ∧ Invalid_requestor! = Copy_check_out_invalid_requestor)

D1_1_2_Check_borrower

⇧ D1_1_Check_copy_check_out (D_borrower_id?, Invalid_borrower!)
D_library_state (Ξ D_is_id_of_borrower)
Valid_borrower_copy_check_out! : 𝔹

D_borrower_id? ➲ Valid_borrower_copy_check_out! ⊗ Invalid__borrower!
(D_borrower_id? ∈ dom D_is_id_of_borrower ∧ Valid_borrower_copy_check_out! = True)
∨ (D_borrower_id? ∉ dom D_is_id_of_borrower ∧ Invalid_borrower! = Copy_check_out_invalid_borrower)

D1_1_3_Check_max_copy

⇧ D1_1_Check_copy_check_out (D_borrower_id?, Within_limit!, Over_limit!)
⇨ D1_1_2_Check_borrower (Valid_borrower_copy_check_out?)
D_library_state (Ξ D_currently_checks_out, Ξ Max_copy)

D_borrower_id? ⊛ Valid_borrower_copy_check_out?➲ Within_limit! ⊗ Over_limit!
((# D_currently_checks_out (| {D_borrower_id?} |) < Max_copy) ∧ Within_limit! = True)
∨ ((# D_currently_checks_out (| {D_borrower_id?} |) ≥ Max_copy) ∧ Over_limit! = Copy_check_out_over_limit)

D1_1_4_Check_copy

⇧ D1_1_Check_copy_check_out (D_copy_id?, Valid_copy!, Invalid_copy!)
D_library_state (Ξ D_is_id_of_copy)

D_copy_id? ➲ Valid_copy! ⊗ Invalid_copy!
(((D_is_id_of_copy (D_copy_id?)).D_status = Available) ∧ (Valid_copy! = True))
∨ (((D_is_id_of_copy (D_copy_id?)).D_status ≠ Available) ∧ (Invalid_copy! = Copy_check_out_invalid_copy))

Chapter 6 Software design verification technique

6.1 Overview

This chapter presents a technique for verifying software design specifications. The software design specification produced must be verified (informally and formally proved) against both the designers' design decisions and software requirements specification, and also it must be verified that it is internally consistent.

Section 6.2 gives the overview of the proposed software design verification technique. Section 6.3 describes how to write a schema to define the relationships between the abstract state space and the concrete state space. Section 6.4 demonstrates how to formally prove that the initial concrete state satisfies the initial abstract state. Finally, section 6.5 describes how to formally prove that the concrete operations implement the abstract operations.

6.2 Overview of the proposed software design verification technique

Software design verification is the process of evaluating that the software design specification satisfies the software requirements specification. The proposed software design specification technique, which will be described in the present chapter, is similar to the proposed software requirements specification technique already described in chapter 4. The verification technique for proving the consistency within an SRS, which is described in chapter 4, can also be applied for proving the consistency within the SDS. Therefore, we will take it for granted by not showing the above mentioned proofs for the SDS. However, we need to verify that the SDS produced satisfies the corresponding SRS. In this chapter, this proof will be discussed.

The overview of the proposed software design verification technique is shown in Figure 6-1 (bold lines represent the areas of applications of the technique). Therefore, the technique covers the following steps: 2.3, 3.1, 3.2, and 3.4. In this chapter, only step 3.2 will be explained; steps 2.3, 3.1, and 3.4 are similar to steps 2.2, 3.1, and 3.3 of the software requirements verification technique accordingly. However, the complete software design verification requires all the steps shown in Figure 6-1.

The steps shown in Figure 6-1 are explained as follows:

Step 1.1

Informal proof is applied to check the informal design specification against the designers' design decisions.

Step 1.2

Informal proof is applied to check the informal design specification against the informal requirement specification.

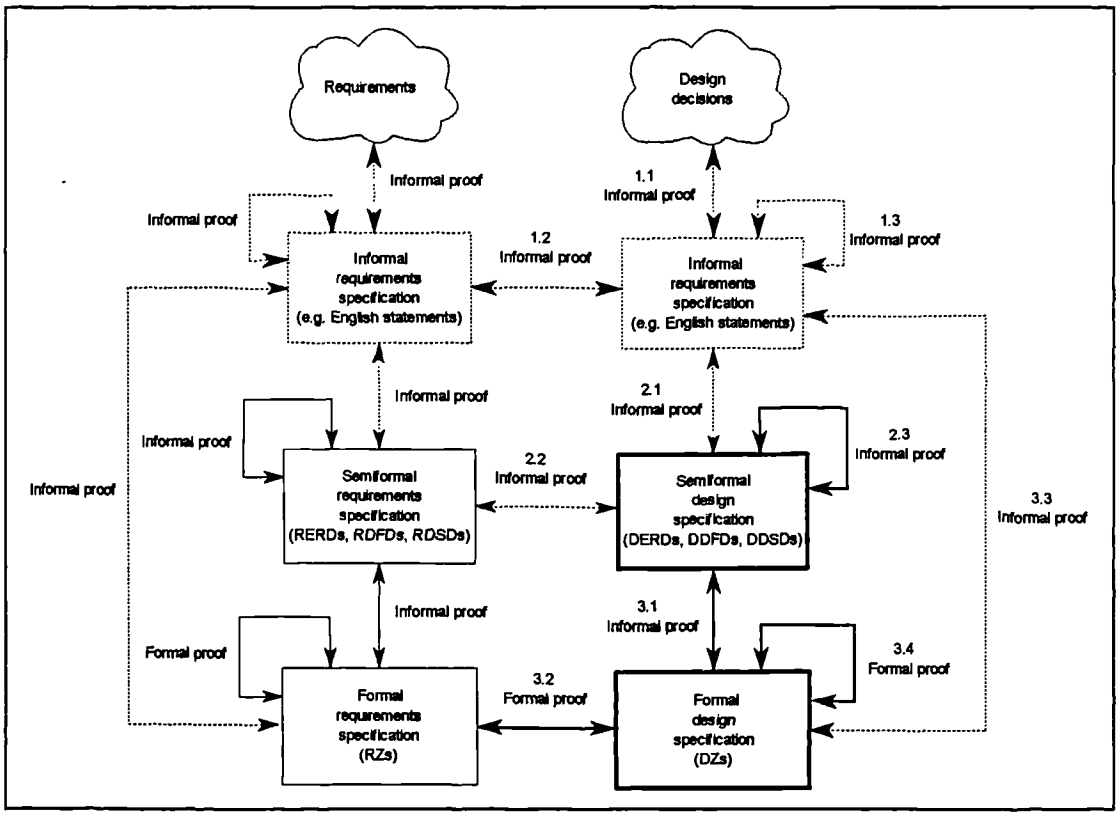


Figure 6-1: Software design verification (bold lines represent areas of application of the proposed technique)

Step 1.3

Informal proof is applied to check the consistency within the informal design specification itself.

Step 2.1

Informal proof is applied to check the semiformal design specification against the informal design specification.

Step 2.2

Informal proof is applied to check the semiformal design specification against the semiformal requirements specification.

Step 2.3

Informal proof is applied to check the consistency within the semiformal design specification itself.

Step 3.1

Informal proof is applied to check the formal design specification against the semiformal design specification.

Step 3.2

Formal proof is applied to check the formal design specification against the formal requirements specification.

Step 3.3

Informal proof is applied to check the formal design specification against the informal design specification.

Step 3.4

Formal proof is applied to mathematically prove the consistency within the formal design specification itself.

6.3 Mapping between the abstract state space and the concrete state space

To prove that the formal design specification satisfies the formal requirements specification, first of all the relationships between the abstract state space and the concrete state space must be identified. In other words, it must be possible to define each abstract data structure in terms of the concrete data structure(s).

Jones [48] uses *retrieving functions* to document the relationships between the abstract state space and the concrete state space. The retrieving functions are used because it is likely that the relationships between the concrete state space and the abstract state space are many-to-one relationships. This idea is also shared by many others, for example Potter et. all [74], Diller [22]. The retrieving functions map the concrete state space into the abstract state space.

In this thesis, we follow what seems to be normal practice for Z practitioners by recording the relationships between the abstract state space and the concrete state space in a schema. Concerning the library system, we write a schema "Retrieve" to document the relationships between the abstract state space "Library_state" and the concrete state space "D_library_state". Each abstract data structure defined in the abstract state space must be mapped into the concrete data structure(s) defined in the concrete state space; and these relationships are documented in the schema "Retrieve".

In this thesis, the symbol $\approx$ is used for defining the mapping between the abstract data structure and its implementation. An abstract data structure is defined on the left hand side of the symbol and its implementation is defined on the right hand side.

For example, the abstract data structure "Borrowers" is implemented by the concrete data structure "D_borrowers" can be written as

$$Borrowers \approx D\_borrowers.$$

The abstract data structure "Borrowers" is defined, in chapter 3, as

$$Borrowers : \mathbb{P}\ PERSON$$

and the abstract data type "PERSON" is defined as a basic type.

The concrete data structure "D_borrowers" is defined, in chapter 5, as

$$D\_borrowers : \mathbb{P}\ D\_BORROWER$$

and the concrete data type "D_BORROWER" is defined as a schema type.

The data type "D_BORROWER" is more concrete than the data type "PERSON"; the data type "PERSON" has no details whereas the data type "D_BORROWER" gives more details.

It is implicit in this use of ≈ that the mapping from the right hand side to the left hand side is a mapping between two sets, and is intended to be well defined, e.g. every element of the set "Borrowers" can be mapped into a corresponding element of the set "D_borrowers".

From chapter 3, the abstract state space "Library_state" is defined as follows.

Library_state

Person_state
Author_state
Subject_state
Book_state
Copy_state
Writes : Library_book_authors •<<→>>• Library_books
Is_a_subject_of : Library_book_subjects •<<→>>• Library_books
Is_a_copy_of : Library_copies •<<→>• Library_books
Currently_checks_out : Borrowers o<→>>• Checked_out_copies
Max_copy : $\mathbb{N}_1$

∀b : dom Currently_checks_out • # Currently_checks_out (| {b} |) ≤ Max_copy

From chapter 5, the concrete state space "D_library_state" is defined as follows.

D_library_state

D_person_state
D_author_state
D_subject_state
D_book_state
D_copy_state
D_writes : D_library_book_author_ids •<<→>>• D_library_book_ids
D_is_a_subject_of : D_library_book_subject_ids •<<→>>• D_library_book_ids
D_is_a_copy_of : D_library_copy_ids •<<→>• D_library_book_ids
D_currently_checks_out : D_borrower_ids o<→>>o D_library_copy_ids
Max_copy : $\mathbb{N}_1$

∀Copy_id : ran D_currently_checks_out •
    D_is_id_of_copy (Copy_id) = Checked_out
∀Borrower_id : dom D_currently_checks_out •
    # D_currently_checks_out (| {Borrower_id} |) ≤ Max_copy

Now, we define the schema "Retrieve" as follows.

Retrieve

Library_state
D_library_state

Borrowers ≈ D_borrowers
Staff ≈ D_Staff
Users ≈ D_borrowers ∪ D_staff
Library_book_authors ≈ D_library_book_authors
Library_book_subjects ≈ D_library_book_subjects

```
Library_books ≈ D_library_books
Library_copies ≈ D_library_copies
Available_copies ≈ { Copy : D_library_copies | Copy.D_status = Available }
Check_out_copies ≈ { Copy : D_library_copies | Copy.D_status = Checked_out }
Writes ≈ ⋃ {Author_id : dom D_writes •
        {D_is_id_of_author (Author_id)} ⤔
        {Book_id : D_writes (| {Author_id} |) • D_is_id_of_book (Book_id)}}
Is_a_subject_of ≈ ⋃ {Subject_id : dom D_is_a_subject_of •
        {D_is_id_of_subject (Subject_id)} ⤔
        {Book_id : D_is_a_subject_of (| {Book_id} |) •
        D_is_id_of_book (Book_id)}}
Is_a_copy_of ≈ {Copy_id : dom D_is_a_copy_of •
        (D_is_id_of_copy (Copy_id),
        D_is_id_of_book (D_is_a_copy_of (Copy_id)))}
Currently_checks_out ≈ {Copy_id : ran D_currently_checks_out •
        ( D_is_id_of_borrower (D_currently_checks_out ^-1 (Copy_id),
        D_is_id_of_copy (Copy_id)))}
```

The abstract data structure "Borrowers" is implemented as the concrete data structure "D_borrowers". The abstract data structure "Staff" is implemented as the concrete data structure "D_staff". The abstract data structure "Users" is not directly implemented in the concrete state space, however the abstract data structure "Users" can be indirectly derived as "D_borrowers ∪ D_staff".

The abstract data structure "Library_book_authors" is implemented as the concrete data structure "D_library_book_authors".

The abstract data structure "Library_book_subjects" is implemented as the concrete data structure "D_library_book_subjects".

The abstract data structure "Library_books" is implemented as the concrete data structure "D_library_books".

The abstract data structure "Library_copies" is implemented as the concrete data structure "D_library_copies". The abstract data structures "Checked_out_copies" and "Available_copies" *are not directly implemented. However the abstract data structure* "Checked_out_copies" can be indirectly derived as

"{ Copy : D_library_copies | Copy.D_status = Available }";

and the abstract data structure "Available_copies" can be indirectly derived as

"{ Copy : D_library_copies | Copy.D_status = Checked_out }".

The abstract data structure "Writes" is defined in the abstract state space as a many-to-many and mandatory-mandatory relationship from "Library_book_authors" to Library_books"; the concrete data structure "D_writes" is defined in the concrete state space as a many-to-many and mandatory-mandatory relationship from "D_library_author_ids" to "D_library_book_ids". Therefore, "Writes" is not directly implemented as "D_writes" but it can be indirectly derived as

"⋃ {Author_id : dom D_writes • {D_is_id_of_author (Author_id)} ⤔
        {Book_id : D_writes (| {Author_id} |) • D_is_id_of_book (Book_id)}}".

The abstract data structure "Is_a_subject_of" is not directly implemented in the concrete state space, but it can be indirectly derived as

"$\bigcup$ {Subject_id : dom D_is_a_subject_of • {D_is_id_of_subject (Subject_id)} $\mapsto$
{Book_id : D_is_a_subject_of (| {Book_id} |) • D_is_id_of_book (Book_id)}}".

The abstract data structure "Is_a_copy_of" is not directly implemented in the concrete state space, but it can be indirectly derived as

"{Copy_id : dom D_is_a_copy_of •
(D_is_id_of_copy (Copy_id), D_is_id_of_book (D_is_a_copy_of (Copy_id)))}".

The abstract data structure "Currently_checks_out" is not directly implemented in the concrete state space, but it can be indirectly derived as

"{Copy_id : ran D_currently_checks_out •
(D_is_id_of_borrower (D_currently_checks_out $^{-1}$ (Copy_id),
D_is_id_of_copy (Copy_id)))}".

Therefore, we have shown that every abstract data structure defined in the abstract state space "Library_state" can be mapped into the concrete data structure(s) defined in the concrete state space "D_library_state".

6.4 Proving that the initial concrete state satisfies the initial abstract state

The next thing that needs to be proved is proving that, given the schema "Retrieve", the initial concrete state is the implementation of the initial abstract state.

For example, in the initial abstract state "Init_library_state", the abstract data structure "Borrowers" is defined as an empty set. According to the schema "Retrieve", "Borrowers" is implemented as "D_borrowers", therefore "D_borrowers" must be defined as an empty set. As defined in the initial concrete state, "D_borrowers" is defined as an empty set, therefore the initial concrete state "D_borrowers" satisfies its initial abstract state. This proof can be shown as follows.

Borrowers	$\approx$ D_borrowers	[from "Retrieve"]
	$\approx \varnothing$	[from "D_Init_library_state"].

In the initial abstract state, the abstract data structure "Writes" is defined to be empty. Therefore, the proof is required to show that

$\bigcup$ {Author_id : dom D_writes • {D_is_id_of_author (Author_id)} $\mapsto$
{Book_id : D_writes (| {Author_id} |) • D_is_id_of_book (Book_id)}}

is also empty. Since the concrete data structure "D_writes" is defined to be empty in the initial concrete state space, so is the set above.

6.5 Proving that the concrete operations implement the abstract operations

To show that the concrete operations correctly implement the abstract operations, we need to prove that each concrete operation and its corresponding abstract operation perform the same operation(s) and produce the same result(s).

For example, we need to prove that the concrete operation "D1_1_1_Check_requestor" and the abstract operation "R_1_1_1_Check_requestor" perform the same operations and produce the same results.

As defined in the abstract operation "R_1_1_1_Check_requestor", the output "Valid_requestor!" is produced if

$$\text{Requestor?} \in \text{Staff}$$

According to the concrete operation "D1_1_1_Check_requestor", the output "Valid_requestor!" is produced if

$$\text{D\_requestor\_id?} \in \text{D\_staff\_ids}$$

Therefore, we must prove that the latter condition and the former condition are in fact state the same thing. From the latter predicate

$$\text{D\_requestor\_id?} \in \text{D\_staff\_ids}$$

we can derive that

$$\text{D\_this\_staff} \in \text{D\_staff}$$

where $\text{D\_this\_staff} = \text{D\_is\_id\_of\_staff}(\text{D\_requestor\_id?})$.

Since

$$\text{Requestor?} \approx \text{D\_is\_id\_of\_staff}(\text{D\_requestor\_id?})$$

and

$$\text{Staff} \approx \text{D\_staff}$$

Therefore, we can conclude that

$$\text{Requestor?} \in \text{Staff}$$

As defined in the abstract operation "R_1_1_1_Check_requestor", the output "Invalid_requestor!" is produced if

$$\text{Requestor?} \notin \text{Staff}$$

According to the concrete operation "D1_1_1_Check_requestor", the output "Invalid_requestor!" is produced if

$$D\_requestor\_id? \notin D\_staff\_ids$$

Therefore, we must prove that the latter condition and the former condition are in fact state the same thing. From the latter predicate

$$D\_requestor\_id? \notin D\_staff\_ids$$

we can derive that

$$D\_this\_staff \notin D\_staff$$

where D_this_staff = D_is_id_of_staff (D_requestor_id?).

Since

$$Requestor? \approx D\_is\_id\_of\_staff\ (D\_requestor\_id?)$$

and

$$Staff \approx D\_staff$$

Therefore, we can conclude that

$$Requestor? \notin Staff$$

Chapter 7 Requirements specification based software testing

7.1 Overview

This chapter describes a technique for generating test requirements for acceptance and system testing from the formal requirements specification.

Section 7.2 gives a brief overview of software testing. In section 7.3, the benefits that formal specifications and software testing bring to one another are pointed out. Section 7.4 gives an overview of the proposed requirements specification based software testing technique. Then, the techniques for deriving test requirements for testing the critical requirements and for testing the operations of the system are explained in section 7.5 and 7.6 accordingly.

7.2 Software testing

Myers [65] defines *testing* as the process of executing a program or system with the intent of finding errors. Hetzel [37] defines *testing* as any activity aimed at evaluating an attribute or capability of a program or system and determining that it meets its required results. In the IEEE standard glossary of software engineering terminology [42], *testing* is defined as the process of exercising or evaluating a system or system component by manual or automated means to verify that it satisfies specified requirements and actual results.

Software testing can be classified into four levels as follows:

1) *Unit testing*

   A unit is the smallest testable piece of software. Unit testing is conducted to show that the unit does not satisfy its functional specification [5].

2) *Integration testing*

   Integration testing is an orderly progression of testing in which software elements, hardware elements, or both are combined and tested until the entire system has been integrated [42]. Integration testing is carried out to test interfaces and ensure that units are communicating as expected. It is done to reveal the incorrectness or inconsistency of the integrated units.

3) *System testing*

   System testing is the process of testing an integrated hardware and software system to verify that the system meets its specified requirements [42]. System testing concerns issues or behaviour that can be exposed by only testing the entire system or a major part of it [5].

4) *Acceptance testing*

Acceptance testing is conducted to determine whether or not a system satisfies its acceptance criteria and to enable the customer to determine whether or not to accept the system [42].

7.3 Formal specifications and software testing

Specifications of both software requirements and design must be used as the basis of quality assurance in some form [11]. Specifications can be used for establishing test requirements (conditions for which we must test [78]), selecting test cases, and evaluation of test results [40].

Formal specifications and formal method have been introduced into the software development process to increase confidence in the software produced. The use of formal specifications and formal method allows early detection of errors and inconsistencies during requirements analysis and design phases rather than late detection during the program testing phase. Theoretically, given a formal specification and a rigorous program development method it is possible to prove that the program produced meets its specification. However, with the current state of art this still can not be guarantee. If the proof is done by hand, there is still room for error [36]. Hence software testing is still necessary.

There are a lot of benefits that formal specifications and software testing bring to one another [11], for example:

1) A formal specification is a valuable source for the systematic derivation of test requirements and test cases. Formal specifications have the ability to precisely describe test requirements and test cases. Formal specification can also be used to validate the test results against the specification.
2) The process of generating test requirements and test cases from a formal specification in turn can assist validation of the formal specification.

7.4 Overview of the proposed technique

This chapter presents a technique for generating test requirements from the formal requirements specification developed as described in chapter 3. The technique concentrates on defining test requirements for acceptance and system testing.

Test requirements can be derived from the formal requirements specification. Test requirements produced can be classified into two sets:

1) Test requirements for testing the critical requirements of the system

   Test requirements for testing the critical requirements of the system are derived from the state space schemas of the formal requirements specification.

2) Test requirements for testing the operations of a system

   Test requirements for testing the operations of the system are derived from the operation schemas of the formal requirements specification.

Once test requirements have been defined, each test requirement has to be instantiated by a collection of test cases which satisfy the preconditions belonging to that test requirement.

7.5 Deriving test requirements for testing the critical requirements of the system

As mentioned in chapter 3 the critical requirements of the system are the requirements that must hold in every state. The critical requirements are very crucial for the system to be valid and they must be tested thoroughly. It is suggested by Hayes in [36] that critical requirements test will be used to check the critical requirements initially and then after every operation performed on the system during testing. To test the critical requirements of the system, one or a set of programs is written to check whether these critical requirements hold.

Test requirements for testing the critical requirements of the system are derived from the state schema(s) of the formal requirements specification of the system.

Concerning the library system, the critical requirements of the system are defined in these state schemas: "Person_state", "Copy_state", and "Library_state". The other three state schemas, "Author_state", "Subject_state", and "Book_state", don't contain any critical requirements.

The critical requirements of the system derived from the three state schemas mentioned above are:

(1) <Borrowers, Staff> partition Users
(2) <Checked_out_copies, Available_copies> partition Library_copies
(3) dom Writes = Library_book_authors
(4) ran Writes = Library_books
(5) dom Is_a_subject_of = Library_book_subjects
(6) ran Is_a_subject_of = Library_books
(7) dom Is_a_copy_of = Library_copies
(8) ran Is_a_copy_of = Library_books
(9) $\forall c$: Library_copies • # Is_a_copy_of (| {c} |) = 1
(10) dom Currently_checks_out $\subseteq$ Borrowers
(11) ran Currently_checks_out = Checked_out_copies
(12) $\forall c$: Checked_out_copies • # Currently_checks_out $^{-1}$ (| {c} |) = 1
(13) Max_copy > 0
(14) $\forall b$: dom Currently_checks_out • # Currently_checks_out (| {b} |) $\leq$ Max_copy

The test requirement (1) is derived from the state schema "Person_state". It states that we must test that: the entities "Borrowers" and "Staff" are disjoint (a person cannot be a borrower and staff at the same time); and the entity "Users" is a union of the entities "Borrowers" and "Staff". Therefore, the test requirement (1) can be split into two test requirements as follows.

(1.1) Borrowers $\cap$ Staff = $\varnothing$

(1.2) Users = Borrowers $\cup$ Staff

Even though according to the formal requirements specification of the library system both test requirements (1.1) and (1.2) are required to be tested, only test requirement (1.1) can be carried out during the program testing. The test requirements (1.2) cannot be tested by program testing. According to the formal design specification of the library system, each borrower as well as each staff is assigned an id and the same id cannot appear in both relations (databases) "D_borrowers" and "D_staff". Therefore, to test the test requirement (1.1) we need to write a program to check that the same id cannot be in both relations (databases).

The test requirement (2) is derived from the state schema "Copy_state". It is similar to the test requirement (1) above.

The test requirements (3) and (4) are derived from the state schema "Library_state". The relationship "Writes" is defined in the schema "Library_state" as a many-to-many and mandatory-mandatory relationship from the entity "Library_book_authors" to the entity "Library_books". Therefore there are two test requirements that must be tested: all authors of the library books must participate in the relationship "Writes" (dom Writes = Library_book_authors); all books in the library must participate in the relationship "Writes" (ran Writes = Library_books).

According to the formal design specification, the relationship "Writes" is implemented by the relation (database) "D_writes" which has two attributes author id and book id. The entity "Library_book_authors" is implemented by the relation "D_library_book_authors" and the entity "Library_books" is implemented by the relation "D_library_books". Therefore we need to write a program to check that all author ids of the relation "D_library_book_authors" and the relation "D_writes" are exactly the same and also all book ids of the relation "D_library_books" and the relation "D_writes" are exactly the same.

The test requirements (5) - (8) are derived from the state schema "Library_state" and are similar to the test requirements (3) - (4) just mentioned.

The test requirement (9) is derived from the state schema "Library_state". It states that each copy of the library can participate only once in the relationship "Is_a_copy_of"; or in other words, a copy is belong to one an only one particular book.

The test requirements (10) - (12) are derived from the state schema "Library_state" and are similar to the test requirements (7) - (9).

The test requirement (13) is derived from the state schema "Library_state" and can be easily checked.

The test requirement (14) is derived from the state schema "Library_state". We need to write a program to check that the number of copies borrowed by each borrower is less than or equal to the predefined number of copies allowed (Max_copy).

7.6 Deriving test requirements for testing the operations of the system

Test requirements for testing the operations of the system are derived from the operation schemas of the formal requirements specification. The cause-effect method is used to partition the input space based on equivalence classes of the output space [11, 65].

The schema which defines the context RDFD is used as the starting point. All inputs and outputs of the system are explicitly defined in this schema. Therefore the test requirement for each distinct output can be easily identified.

Concerning the library system, the schema "R0_Library" is used as the starting point to derive test requirements for testing the operations of the system. From the schema "R0_Library", all test requirements can be identified from the output declaration statements in the declaration part of the schema.

The total number of test requirements as identified from the schema "R0_Library" are 42 test requirements. The number of test requirements for each output are shown in Table 7-1.

Output	Number of test requirements
1. Valid_copy_check_out!	1
2. Invalid_copy_check_out!	11
3. Valid_copy_return!	1
4. Invalid_copy_return!	3
5. Valid_copy_add!	1
6. Invalid_copy_add!	3
7. Valid_copy_remove!	1
8. Invalid_copy_remove!	3
9. List_of_books_by_author!	1
10. Invalid_by_author_enquiry!	3
11. List_of_books_by_subject!	1
12. Invalid_by_subject_enquiry	3
13. List_of_copies_by_borrower!	1
14. Invalid_by_borrower_enquiry	5
15. Last_borrower_info!	1
16. Invalid_last_borrower_enquiry!	3
Total	42

Table 7-1: Number of test requirements of each output

The output "Valid_copy_check_out!" as declared in the specification contains no condition, therefore only one test requirement is derived.

The output "Invalid_copy_check_out!" is defined in the schema "R0_library" as

Invalid_copy_check_out! : (Invalid_requestor : MSG ⓥ

(Invalid_borrower : MSG ⊗ Over_limit : MSG) ⓥ Invalid_copy : MSG

The output "Invalid_copy_check_out!" as declared in the specification contains disjunction and exclusive disjunction notations, therefore in fact there are 11 combinations for this output as follows.

1) Invalid_requestor!

2) Invalid_borrower!

3) Over_limit!

4) Invalid_copy!

5) Invalid_requestor! ⓐ Invalid_borrower!

6) Invalid_requestor! ⓐ Over_limit!

7) Invalid_requestor! ⓐ Invalid_copy!

8) Invalid_borrower! ⓐ Invalid_copy!

9) Over_limit! ⓐ Invalid_copy!

10) Invalid_requestor! ⓐ Invalid_borrower! ⓐ Invalid_copy!

11) Invalid_requestor! ⓐ Over_limit! ⓐ Invalid_copy!

Even though there are four errors concerning the output "Invalid_copy_check_out", all four errors cannot happen in the same case. This is because the error "Invalid_borrower!" and the error "Over_limit!" cannot happen together.

The number of test requirements for other outputs can be formulated in a similar way.

Then, test requirement specifications for each test requirement can be derived from the operation schemas by following these steps.

1) Find the operation schema which generates that output.
2) From that schema, identify the inputs and postconditions. The inputs are defined on the left hand side of the input/output data flow relation statement. The postconditions are derived from the predicates of that schema. The inputs identified may be either the inputs from the external entities or the internal inputs generated within the system. The preconditions are derived from the preconditions of the processes that produce those internal inputs and from the predicates of that schema. The inputs required in order to produce the required output are the inputs passed from the external entities.
3) Write test requirements specification of that output using the format shown in Table 7-2.

Test requirement n	
Inputs	
Preconditions	
Postconditions	
Outputs	

Table 7-2: Test requirement specification format

A. Test requirement 1

The test requirement specification of the output "Valid_copy_check_out!" can be derived by following the steps given above and will be explained as follows.

1) The operation schema which generates that output is "R1_2_Record_copy_check_out".
2) The input/output data flow relation statement defined in that schema is

Borrower? ⓐ Copy? ⓐ Valid_copy_check_out_check? ➲ Valid_copy_check_out!

The inputs are:

"Borrower?"

"Copy?"

"Valid_copy_check_out_check?"

The inputs "Borrower?" and "Copy?" are the inputs passed from the external entity; whereas the input "Valid_copy_check_out_check?" is the internal data flow. Therefore the preconditions as well as the remaining inputs will be derived from the input "Valid_copy_check_out_check?" as follows.

The input "Valid_copy_check_out_check?" is passed from the schema "R1_1_Check_copy_check_out" and it is declared in that schema as

$$\text{Valid\_copy\_check\_out\_check!} : (\text{Valid\_requestor} : \mathbb{B} \otimes \text{Within\_limit} : \mathbb{B} \otimes \text{Valid\_copy} : \mathbb{B})$$

Therefore, we need in turn to find out the inputs and preconditions of the outputs "Valid_requestor!", "Wihtin_limit!", and "Valid_copy!".

The input of the output "Valid_requestor!" is "Requestor?" and the precondition is defined in the schema "R1_1_1_Check_requestor" as

$$\text{Requestor?} \in \text{Staff}$$

The input of the output "Within_limit!" is "Borrower?" and the preconditions are defined in the schemas "R1_1_2_Check_borrower" and "R1_1_3_Check_max_copy" as

$$(\text{Borrower?} \in \text{Borrowers}) \wedge$$
$$(\#\ \text{Currently\_checks\_out}\ (|\ \{\text{Borrower?}\}\ |) < \text{Max\_copy})$$

The input of the output "Valid_copy!" is "Copy?" and the precondition is defined in the schema "R1_1_4_Check_copy" as

$$\text{Copy?} \in \text{Available\_copies}$$

Therefore, the inputs of the output "Valid_copy_check_out!" are:

Requestor?
Borrower?
Copy?

and the preconditions of the output "Valid_copy_check_out!" are the conjunction of the preconditions above:

$$(\text{Requestor?} \in \text{Staff}) \wedge$$
$$(\text{Borrower?} \in \text{Borrowers}) \wedge$$
$$(\#\ \text{Currently\_checks\_out}\ (|\ \{\text{Borrower?}\}\ |) < \text{Max\_copy}) \wedge$$
$$(\text{Copy?} \in \text{Available\_copies})$$

The postconditions of the output are the predicates of the schema "R1_2_Record_copy_check_out" as follows.

$$\text{Available\_copies}' = \text{Available\_copies} \setminus \{\text{Copy?}\}$$

Checked_out_copies ' = Checked_out_copies ∪ {Copy?}
Currently_checks_out ' = Currently_checks_out ∪ {Borrower? ↦ Copy?}
Valid_copy_check_out! = Copy_check_out_ok

3) Then the test requirement specification of the output "Valid_copy_check_out!" can be written as follows.

Test requirement 1	
Inputs	Requestor? Borrower? Copy?
Preconditions	Requestor? ∈ Staff Borrower? ∈ Borrowers # Currently_checks_out (| {Borrower?} |) < Max_copy Copy? ∈ Available_copies
Postconditions	Available_copies ' = Available_copies \ {Copy?} Checked_out_copies ' = Checked_out_copies ∪ {Copy?} Currently_checks_out ' = Currently_checks_out ∪ {Borrower? ↦ Copy?} Valid_copy_check_out! = Copy_check_out_ok
Outputs	Valid_copy_check_out!

Each test requirement leads to a number of test cases. For example, from the test requirement 1 given above, a number of test cases can be produced. However, all test cases which belong to a particular test requirement must satisfy the specification of that test requirement. Various testing techniques such as boundary-value analysis, category-partition, and/or branch coverage can be applied to generate test cases from a test requirement. However, the detailed consideration of how the different test cases can be derived from the test requirment is beyond the scope of this thesis.

B. Test requirement 2

The test requirement specification of the output "Copy_check_out!.Invalid_requestor" can be derived by following the steps given above. The result is shown in the following table.

Test requirement 2	
Inputs	Requestor? Borrower? Copy?
Preconditions	Requestor? ∉ Staff Borrower? ∈ Borrowers # Currently_checks_out (| {Borrower?} |) < Max_copy Copy? ∈ Available_copies
Postconditions	Invalid_requestor! = Copy_check_out_invalid_borrower
Outputs	Copy_check_out!.Invalid_requestor

C. Test requirement 3

The test requirement specification of the output "Copy_check_out!.Invalid_borrower" is shown in the following table.

Test requirement 3	
Inputs	Requestor? Borrower? Copy?
Preconditions	Requestor? ∈ Staff Borrower? ∉ Borrowers # Currently_checks_out (| {Borrower?} |) < Max_copy Copy? ∈ Available_copies
Postconditions	Invalid_requestor! = Copy_check_out_invalid_borrower
Outputs	Copy_check_out!.Invalid_borrower

D. Test requirement 4

The test requirement specification of the output "Copy_check_out!.Over_limit" is shown in the following table.

Test requirement 4	
Inputs	Requestor? Borrower? Copy?
Preconditions	Requestor? ∈ Staff Borrower? ∈ Borrowers # Currently_checks_out (| {Borrower?} |) ≥ Max_copy Copy? ∈ Available_copies
Postconditions	Invalid_requestor! = Copy_check_out_over_limit
Outputs	Copy_check_out!.Over_limit

E. Test requirement 5

The test requirement specification of the output "Copy_check_out!.Invalid_copy" is shown in the following table.

Test requirement 5	
Inputs	Requestor? Borrower? Copy?
Preconditions	Requestor? ∈ Staff Borrower? ∈ Borrowers # Currently_checks_out (| {Borrower?} |) < Max_copy Copy? ∉ Available_copies
Postconditions	Invalid_requestor! = Copy_check_out_invalid_copy
Outputs	Copy_check_out!.Invalid_copy

F. Test requirement 6

The test requirement specification of the combined output "Copy_check_out!.Invalid_requestor Ⓐ Copy_check_out!.Invalid_borrower" is shown in the following table.

Test requirement 6	
Inputs	Requestor? Borrower? Copy?
Preconditions	Requestor? ∉ Staff Borrower? ∉ Borrowers # Currently_checks_out (| {Borrower?} |) < Max_copy Copy? ∈ Available_copies
Postconditions	Invalid_requestor! = Copy_check_out_invalid_requestor Invalid_borrower! = Copy_check_out_invalid_borrower
Outputs	Copy_check_out!.Invalid_requestor Ⓐ Copy_check_out!.Invalid_borrower

G. Test requirement 7

The test requirement specification of the combined output "Copy_check_out!.Invalid_requestor Ⓐ Copy_check_out!.Over_limit" is shown in the following table.

Test requirement 7	
Inputs	Requestor? Borrower? Copy?
Preconditions	Requestor? ∉ Staff Borrower? ∈ Borrowers # Currently_checks_out (| {Borrower?} |) ≥ Max_copy Copy? ∈ Available_copies
Postconditions	Invalid_requestor! = Copy_check_out_invalid_requestor Invalid_borrower! = Copy_check_out_over_limit
Outputs	Copy_check_out!.Invalid_requestor Ⓐ Copy_check_out!.Over_limit

H. Test requirement 8

The test requirement specification of the combined output "Copy_check_out!.Invalid_requestor Ⓐ Copy_check_out!.Invalid_copy" is shown in the following table.

Test requirement 8	
Inputs	Requestor? Borrower? Copy?
Preconditions	Requestor? ∉ Staff Borrower? ∈ Borrowers # Currently_checks_out (| {Borrower?} |) < Max_copy Copy? ∉ Available_copies
Postconditions	Invalid_requestor! = Copy_check_out_invalid_requestor Invalid_borrower! = Copy_check_out_invalid_copy
Outputs	Copy_check_out!.Invalid_requestor Ⓐ Copy_check_out!.Invalid_copy

I. Test requirement 9

The test requirement specification of the combined output "Copy_check_out!.Invalid_borrower Ⓐ Copy_check_out!.Invalid_copy" is shown in the following table.

Test requirement 9	
Inputs	Requestor? Borrower? Copy?
Preconditions	Requestor? ∈ Staff Borrower? ∉ Borrowers # Currently_checks_out (| {Borrower?} |) < Max_copy Copy? ∉ Available_copies
Postconditions	Invalid_requestor! = Copy_check_out_invalid_borrower Invalid_borrower! = Copy_check_out_invalid_copy
Outputs	Copy_check_out!.Invalid_borrower Ⓐ Copy_check_out!.Invalid_copy

J. Test requirement 10

The test requirement specification of the combined output "Copy_check_out!.Over_limit Ⓐ Copy_check_out!.Invalid_copy" is shown in the following table.

Test requirement 10	
Inputs	Requestor? Borrower? Copy?
Preconditions	Requestor? ∈ Staff Borrower? ∈ Borrowers # Currently_checks_out (| {Borrower?} |) ≥ Max_copy Copy? ∉ Available_copies
Postconditions	Invalid_requestor! = Copy_check_out_over_limit Invalid_borrower! = Copy_check_out_invalid_copy
Outputs	Copy_check_out!.Over_limit Ⓐ Copy_check_out!.Invalid_copy

K. Test requirement 11

The test requirement specification of the combined output "Copy_check_out!.Invalid_requestor Ⓐ Copy_check_out!.Invalid_borrower Ⓐ Copy_check_out!.Invalid_copy" is shown in the following table.

Test requirement 11	
Inputs	Requestor? Borrower? Copy?
Preconditions	Requestor? ∉ Staff Borrower? ∉ Borrowers # Currently_checks_out (| {Borrower?} |) < Max_copy Copy? ∉ Available_copies
Postconditions	Invalid_requestor! = Copy_check_out_invalid_requestor Invalid_borrower! = Copy_check_out_invalid_borrower Invalid_borrower! = Copy_check_out_invalid_copy
Outputs	Copy_check_out!.Invalid_requestor Ⓐ Copy_check_out!.Invalid_borrower Ⓐ Copy_check_out.Invalid_copy

L. Test requirement 12

The test requirement specification of the combined output "Copy_check_out!.Invalid_requestor Ⓐ Copy_check_out!.Over_limit Ⓐ Copy_check_out!.Invalid_copy" is shown in the following table.

Test requirement 12	
Inputs	Requestor? Borrower? Copy?
Preconditions	Requestor? ∉ Staff Borrower? ∈ Borrowers # Currently_checks_out (| {Borrower?} |) ≥ Max_copy Copy? ∉ Available_copies
Postconditions	Invalid_requestor! = Copy_check_out_invalid_requestor Invalid_borrower! = Copy_check_out_over_limit Invalid_borrower! = Copy_check_out_invalid_copy
Outputs	Copy_check_out!.Invalid_requestor Ⓐ Copy_check_out!.Over_limit Ⓐ Copy_check_out!.Invalid_copy

Chapter 8 Conclusion

8.1 Overview

In this final chapter, we hope to round off the proceedings by summarizing the proposed techniques, comparing the proposed techniques with related works, and suggesting potential further development.

Section 8.2 summarizes the techniques and points out their novelty as well as their limitations. In section 8.3, we compare the proposed techniques with five prominent related works. Finally, in section 8.4, further development are suggested.

8.2 Summary of the proposed techniques

Specifying, verifying, and testing software requirements and design are very important tasks in the software development process and must be taken seriously. By investing more up-front effort in these tasks, projects will gain the benefits of reduced maintenance costs, higher software reliability, and more user-responsive software [9].

In this thesis, we have proposed very practical and efficient techniques for specifying and verifying software requirements and design and for generating test requirements for acceptance and system testing. The proposed techniques are designed to be applied to any typical business information system (in other words to NH, BA and IN, ST and MT, DA and DE applications, see section 1.5). To demonstrate the practicality of the proposed techniques, the techniques have been thoroughly demonstrated through a simplified library system (an overview of the library system is given in section 1.10).

Good software specifications should have these characteristics: understandable, unambiguous, consistent, correct, complete, verifiable, modifiable, and traceable (see section 1.6). One of the best solutions to achieve such good software specifications is by integrating three categories of software specification languages, namely informal specification languages (e.g. English language), semiformal specification languages (e.g. entity-relationship diagrams, data flow diagrams, and data structure diagrams), and formal specification languages (e.g. Z).

Both the proposed software requirements and design specification techniques emerge from integrating the above mentioned three categories of software specification languages. Even though we suggest using an informal specification language, it has not been discussed in this thesis. For semiformal and formal specification languages (ERDs, DFDs, DSDs, and Z), we adapt and extend them (as explained in chapter 2) so that they are nicely integrated and are capable of producing good software specifications of the type we aim for. Both software requirements and software design are specified in terms of the static and dynamic aspects of a system. The static aspects of a system are specified by using ERDs, DSDs, and Z state specifications whereas the dynamic aspects of a system are specified by using DFDs, DSDs, and Z operation specifications. Even though the same specification languages are used for specifying both software requirements and design, the contents of the two specifications are different. We have demonstrated step by step how to produce software requirements

specifications (in chapter 3) and how to produce software design specifications (in chapter 5). We have also discussed (in section 1.6) and shown (in chapter 3 and 5) the differences between software requirements specifications and software design specifications.

The proposed software requirements verification technique is a *combination of* informal and formal proofs (explained in chapter 4). We have explained thoroughly showing: 1) how to apply informal proofs (informal checking) to check the consistency within the semiformal requirements specification (RERDs, RDFDs, and RDSDs); 2) how to apply informal proofs to check the consistency, correctness, and completeness of the formal requirements specification (RZs) against the semiformal requirements specification (RERDs, RDFDs, and RDSDs); 3) how to apply formal proofs to mathematically prove the consistency of the formal requirements specification (RZs).

The proposed software design verification technique (described in chapter 5) is similar to the proposed software requirements verification technique mentioned above. The same verification technique as the software requirements verification technique is also applied to: 1) informally prove (informally check) the consistency within the semiformal design specification (DERDs, DDFDs, and DDSDs); 2) informally prove the consistency, correctness, and completeness of the formal design specification (DZs) against the semiformal design specification (DERDs, DDFDs, and DDSDs); 3) formally prove the consistency within the formal design specification (DZs). Therefore, we take it for granted by not demonstrating the above mentioned proofs in this thesis. We have, however, explained how to apply formal proofs to mathematically prove that the formal design specification (DZs) satisfies the formal requirements specification (RZs).

Finally, we have demonstrated how to generate test requirements for acceptance and system testing (explained in chapter 7). Such test requirements are generated from the formal requirements specification (RZs). Two sets of test requirements are produced: 1) test requirements for testing the critical requirements of a system; and 2) test requirements for testing the operations of a system.

Although no detailed trials have been carried out to establish if the techniques proposed are effective and useful, a number of correspondents have received the ideas very favourably.

8.2.1 Novelty of the proposed techniques

The proposed techniques, we believe, have many distinctive features as follows.

They cover a wide range of the earlier tasks of the software development process (see Figure 1-1): producing software requirements specifications, verifying software requirements specifications, producing software design specifications, verifying software design specifications, and generating test requirements for acceptance and system testing. The tasks mentioned above are closely related and therefore the techniques for handling those tasks should support one another. The proposed techniques have been carefully developed so that they are fully integrated and support one another nicely. In addition, we have also provided the procedures for managing the proposed techniques. Therefore, as a result we do believe that the practitioners will find the proposed techniques useful, not too difficult, and can be applied in real practices.

Software requirements specifications as well as software design specifications produced by following the proposed techniques have many good characteristics as follows. They are easy to understand; the specifications are specified in a natural language (e.g. English statements) as well as by using graphical models (ERDs, DFDs, and DSDs) which are easy to understand, especially by the end-users. They are unambiguous and verifiable; the specifications are also formally specified in the Z specification language therefore they are precise (unambiguous) and can be proved formally.

The proposed software requirements and design specification techniques integrate one of the most popular formal specification languages, Z, with three of the most well-known semiformal specification languages, ERDs, DFDs, and DSDs. The three diagrams above are used by most of the structured software development methods. Therefore, the proposed techniques, which integrate these four popular specification languages, permit users to continue using available popular specification languages and therefore the users will find that it is easy to learn and use them.

The semiformal specification and the corresponding formal specification can be easily converted from one form to another. Hence, those who produce these specifications will find that the tasks are straight forward and can be easily accomplished. Furthermore, this also allows the parties to communicate effectively, for example the end-users can learn what have been stated in the specifications from the informal and semiformal specifications, while the system analysts can get more details about the specifications from the formal specifications. If the semiformal specification and the corresponding formal specification cannot be easily mapped, it would be difficult producing the specifications as well as communicating their meaning.

The verification techniques proposed in this thesis cover both software requirements and design specifications. In addition, not only the informal proofs are given, but also the formal proofs. Therefore, the practitioners will feel more confident about the software requirements and design specifications produced.

The proposed technique of generating test requirements from the formal requirements specification produces test requirements for acceptance and system testing in which the test requirements are formally specified using the Z specification language. This results in the test requirements specifications which are precise.

8.2.2 Limitations of the proposed techniques

Unfortunately, the proposed techniques also have some drawbacks as follows.

The proposed techniques are only suitable for typical business information systems (NH, BA and IN, ST and MT, DA, and DE applications, see section 1.5). They are not suitable for more complicated systems (HA, AL, ND *applications, see section 1.5), such as* real-time systems.

To produce a software requirements specification as well as a software design specification by following the proposed techniques, three sets of specifications (an informal, semiformal, and formal specifications) must be produced. Therefore the amount of work that has to be done is approximately triple the minimum amount of work normally done.

Although the four specification languages used by the proposed techniques are all well-known, they have been adapted and extended. Even though we tried to keep the changes as minimum as possible, some effort is still required to master them.

Due to the difficulty in specifying ternary relationships (relationships which involving three entities) in formal specifications, the proposed techniques do not suggest how to specify ternary relationships. Therefore, to use the proposed techniques, a ternary relationship must be converted into two or more binary relationships.

In this thesis, some formal proofs have been proposed. Even though the formal proofs proposed have been thoroughly outlined, the proof processes have not yet been fully defined. More rigorous proof processes are still required.

8.3 A comparison with related works

In recent years, the idea of integrating formal specifications with informal and semiformal specifications has become more and more popular. Many ideas (approaches, methods, techniques) for the above subject have already been proposed. In this section, we will compare our techniques with some other notable techniques as follows.

Kung

Kung [53] proposed the conceptual model approach. This approach has the following features: 1) it uses entity-relationship diagrams to depict the static aspects of a system, data flow diagrams to depict dynamic aspects of a system, and a combination of entity-relationship diagrams, data flow diagrams, Petri nets, and relational calculus to describe the process behaviors of a system; 2) it describes how to formally prove the consistency of the process decomposition of DFDs; 3) it briefly discusses on correctness checking of behaviour specifications; 4) it produces an executable specification which can be translated into a Prolog program to simulate the system being modelled.

This approach and our approach are similar in the following aspects. We borrow from this approach the concept of using the data interface operators (see section 2.3.1.5) for describing the relationship among input or output data flows. However, we have modified them (changed the notation and changed the way of drawing it) to gain a better result.

We also adapted an idea of how to apply formal proofs to prove the consistency of the process decomposition (see section 4.5.3). However, our proof method is more rigorous and more complete than the one offered in this approach. Kung has given 4 logical rules whereas we have given 24 logical rules. The main problem of the proof method given in Kung's approach is that his proof does not distinguish between input and output data flows. Therefore, the logical rule I3 given in his approach is incorrect if the data flows are considered to be output data flows; but this logical rule is correct if the data flows are correctly identified, i.e. which one is an input data flow and which one is an output data flow. Unfortunately, the exact interpretation of this rule has not been stated in [53]. This point needs to be explained in more details as follows.

In Kung's paper [53], the logical rule I3 is given as follow.

I3. $\models A \bullet (B \oplus C) \Rightarrow (A \bullet B) \oplus C$

where $\bullet$ is a conjunction;

$\oplus$ is an exclusive disjunction

If data flows A, B, and C are all thought to be output data flows, this rule is false. This is because the left hand side of the rule is then interpreted as A and either B or C, but not both B and C, are produced, whereas the right hand side of the rule is interpreted as either A and B or C, but not both, are produced. In other words, the left hand side of the rule states that two combinations of output data flows can be produced: 1) $A \bullet B$; 2) $A \bullet C$. The right hand side of the rule states that two combinations of output data flows can be produced: 1) $A \bullet B$; 2) C. Therefore the right hand side cannot be inferred from the left hand side, and hence the rule is incorrect.

However, the logical rule I3 is correct if it is given as follow.

I3. $\models A? \bullet (B! \oplus C!) \Rightarrow (A? \bullet B!) \oplus C!$

where ? is used to identify that a data flow is an input data flow;

! is used to identify that a data flow is an output data flow.

This logical rule states that given an input data flow A?, output data flows B! and C!, and A? is in conjunction with the exclusive disjunction of B! and C!, we then can derive either A? and B! or C!, but not both. In other words, the input data flow A? can be combined with the output data flow B! (B! is an output data flow from one process and is passed as an input data flow to another process) and whatever output data flow(s) produced, this output data flow must be in exclusive disjunction with the output data flow C!. This is correct.

This approach and our approach are different in the following major aspects. Some of the diagrams used in both approaches are different. Our approach uses data structure diagrams which is not used in this approach. Kung's approach uses Petri nets which is not used in our approach. This approach concerns only specifying software requirements specifications whereas our approach concerns both software requirements and design specifications.

Fraser et. all

Fraser et. all [30] proposed two approaches for integrating Structured Analysis (SA) with the Vienna Development Method (VDM). The first approach proposes generating the VDM specifications by using Structured Analysis specifications as cognitive guides to develop VDM specifications. The second approach proposes generating the VDM specifications by using a rule-based method. The SA specifications are in forms of data flow diagrams, data dictionary, and the transform descriptions associated with the bottom-level processes which are given as decision tables.

In their first approach, the derivation of VDM specifications from the SA specifications is done by an analyst (in other words is done by a human). In their second approach, the VDM specifications are partially automatically generated. In the second approach, the VDM specifications are derived by mapping decision table process-descriptions into VDM specifications using the decision table conversion rule and the sequence composition rule.

There is one similarity between their approach and our approach: both produce formal specifications for all processes on DFDs.

Their approach and our approach are different in many aspects as follows. Their approach uses two semiformal specification languages, DFDs and decision tables, and data dictionary; whereas our approach uses DFDs, ERDs, and DSDs. The formal specification languages used by the two approaches are different, VDM vs Z. This approach concerns only specifying software requirements specifications whereas our approach concerns both software requirements and design specifications.

Polack et. all

Polack et. all [71, 72, 73] proposed the SAZ method which is a method for integrating SSADM with Z. In the SAZ method, the following Z specifications are generated: the specification of the system state; the specification of critical processing; the specification of selected enquiries. The Z specifications are derived from the products of the SSADM Requirements Analysis and Specification modules, specifically, the logical data model, function definitions, entity life histories, effect correspondence diagrams, enquiry access paths, and I/O structures.

The SAZ method and our method are quite different. Even though both methods derive the Z state specifications from ERDs, the Z state specifications produced from the two methods

are different, for example each relationship shown in the ERD is defined twice (the relationship is defined both ways) in the SAZ method whereas in our method it is defined just once, to define a relationship, in the SAZ method the standard Z relation and function notations are used whereas in our method a relationship is defined by using new proposed relationship notations (see section 2.5.1.1). In the SAZ method only critical functions and selected enquiries are defined in Z specifications whereas in our method all functions are defined in Z specifications. In our method, the Z operation specifications are derived from DFDs; whereas in the SAZ method, the Z operation specifications are derived mainly from ELH diagrams.

Semmens et. all

Semmens et. all [82] proposed a method of using the Z notation together with Yourdon Structured Analysis to produce a requirements specification. In this approach, the formal specifications of the system state are derived from entity-relationship diagrams and the formal specifications of the system operations are derived from data flow diagrams.

This approach and our approach use the two same diagrams, ERDs and DFDs, and the same formal specification language, Z. However, the diagram techniques as well as the Z specification language used by the two approaches are slightly different; in our approach we have modified and extended them whereas this approach uses the standard notations. In our approach, we also use DSDs which is not used in this approach.

Both approaches specify a system in terms of the static and dynamic aspects of a system. However, in this approach, only bottom-level processes on DFDs are given formal specifications. In our approach, all processes on DFDs are given formal specifications.

This approach covers only specifying software requirements specifications whereas our approach covers both software requirements and design specifications.

Randell

Randell [76] proposed an approach for translating a data flow diagram into an outline Z specification and also an approach for generating a data flow diagram from a Z specification.

This approach concerns only combining data flow diagrams and Z. The Z specifications of the state space and operations of a system are derived from the DFDs. However, only bottom-level processes on DFDs are given the Z specifications; there is no Z specifications for higher-level processes.

This approach and our approach are similar in that both of them use DFDs and Z. However, the concepts of DFDs and the methods for generating Z specifications in both approaches are different.

8.4 Further development of the proposed techniques

The proposed techniques have been carefully planned so that they can be extended later on. Some of the potential and useful extensions that can be added to the proposed techniques are as follows.

The proposed techniques should be extended so that they can be applied to real-time systems. Since more and more systems being developed are real-time systems, it is important that the techniques should be able to handle such systems.

A lot of improvements concerning formal proofs are required. One of the most potential benefits of formal proofs which is worth including is symbolic execution.

The following automated tools can be developed and added to the proposed techniques:

1) Automated tools for drawing the modified versions of ERDs, DFDs, and DSDs and also for adding Z specifications can be developed. Even though there are many CASE (Computer Aided Software Engineering) tools available which supports those diagrams, none of them offers exactly the same notations as required. However, it is possible to modify the available CASE tools so that they would be able to support the notations required. It is also possible to extend the available CASE tools to accept Z specifications. These extended CASE tools will help to reduce the amount of time and effort required for the tasks.

2) Automated tools for generating formal specifications from semiformal specifications and vice versa can be developed. Based on the software requirements and design specification techniques proposed, around 60 percent of the formal specifications can be automatically generated from the corresponding semiformal specification, and a complete (100 percent) semiformal specification can be automatically generated from the formal specification.

3) An automated theorem proving tool is required. Theoretically, the formal proofs as suggested in the proposed techniques can be carried out mechanically by an automated theorem proving tool. However, with the current state of the art it would be very expensive and take quite a long time to produce such a tool. However, a symbolic execution tool is more likely (possible) to be produced, and such a tool will be found very useful in verify formal specifications.

4) A rapid prototyping tool, which is an automated tool for translating the formal specification into executable programs to simulate the system being model, can be developed. This tool will provide users with a prototype which they can exercise to see if it meets their functional requirements.

Bibliography

[1] "Problem Set for the Fourth International Workshop on Software Specification and Design," *Proc. 4th International Workshop on Software Specification and Design*, pp. ix-x, April 1986

[2] C. Ashworth and M. Goodland, *SSADM: A Practical Approach*, McGraw-Hill, 1990

[3] G. Babin, F. Lustman, and P. Shoval, "Specification and Design of Transactions in Information Systems: A Formal Approach," *IEEE Transactions on Software Engineering*, vol. 17, no. 8, August 1991

[4] R. Balzer and N. Goldman, "Principles of Good Software Specification and their Implications for Specification Languages," *Proc. IEEE Conference on Specifications of Reliable Software*, pp. 58-67, 1979

[5] B. Beizer, *Software Testing Techniques*, Van Nostrand Reinhold, 1990

[6] D. Bell, I. Morrey, and J. Pugh, *Software Engineering: A Programming Approach*, Prentice-Hall, 1987

[7] B. W. Boehm, "Software Engineering," *IEEE Transactions on Computers*, vol. C-25, no. 12, pp. 35-50, December 1976.

[8] B. W. Boehm, *Software Engineering Economics*, Prentice-Hall, 1981

[9] B. W. Boehm, "Verifying and Validating Software Requirements and Design Specifications," *IEEE Software*, vol. 1, no. 1, pp. 75-88, January 1984

[10] S. H. Caine and K. Gordon, "PDL - A Tool for software Design," *Proc. National Computer Conference*, 1975

[11] D. Carrington and P. Stocks, "A Tale of Two Paradigms: Formal Methods and Software Testing," *Proc. 8th Annual Z User Meeting*, Cambridge, June 1994, Springer-Verlag, 1994

[12] W. Chantatub and M. Holcombe, "Software Testing Strategies for Software Requirements and Design," *Proc. 2st European International Conference on Software Testing Analysis & Review*, Brussels, October 1994

[13] P. P. Chen, "The Entity-Relationship Model - Towards a Unified View of Data," *ACM Transactions on Database Systems*, vol. 1, no. 1, pp. 9-36, March 1976

[14] P. P. Chen (Ed.), *Proc. 1st Int. Conference on Entity-Relationship Approach to Systems Analysis and Design*, 1979, North Holland, 1980

[15] J. S. Collofello and L. B. Balcom, "A Proposed Causative Software Error Classification Scheme," *Proc. National Computer Conference*, 1985

[16] A. Coombes and J. McDermid, "A Tool for Defining the Architecture of Z Specifications," *Proc. 5th Annual Z User Meeting*, Oxford, December 1990, Springer-Verlag, 1991

[17] C. J. Date, *An Introduction to Database Systems*, 4th ed., Addison-Wesley, 1986

[18] A. M. Davis, E. H. Bersoff, and E. R. Comer, "A Strategy for Comparing Alternative Software Development Life Cycle Models," *IEEE Transactions on Software Engineering*, vol. 14, no. 10, pp. 1453-1461, October 1988

[19] A. M. Davis, *Software Requirements: Analysis & Specification*, Prentice-Hall, 1990

[20]. T. Demacro, *Structured Analysis and System Specification*, Yourdon Press, 1978

[21] M. S. Deutsch, *Software Verification and Validation: Realistic Project Approaches*, Prentice-Hall, 1982

[22] A. Diller, *Z: An Introduction to Formal Methods*, John Wiley & Sons, 1991

[23] E. Dubois and A. Lamsweerde, "Making Specification Processes Explicit," *Proc. 4th International Workshop on Software Specification and Design*, pp. 169-177, CS Press, 1987

[24] R. H. Dunn, *Software Defect Removal*, McGraw-Hill, 1984

[25] N. Fenton, "Software Measurement: Why a Formal Approach?," *Proc. BCS-FACS Workshop on Formal Aspects of Measurement*, South Bank University, London, May 1991, pp. 3-27, Springer Verlag, 1992

[26] N. Fenton and G. Hill, *Systems Construction and Analysis: A Mathematical and Logical Framework*, McGraw-Hill, 1993

[27] S. Fickas, "Automating the Analysis Process: An Example," *Proc. 4th International Workshop on Software Specification and Design*, pp. 58-67, CS Press, 1987

[28] A. S. Fisher, *CASE: Using Software Development Tools*, John Wiley & Sons, 1988

[29] M. Flavin, *Fundamental Concepts of Information Modeling*, Yourdon Press, 1981

[30] M. D. Fraser, K. Kumar, and V. K. Vaishnavi, "Informal and Formal Requirements Specification Languages: Bridging the Gap," *IEEE Transactions on Software Engineering*, vol. 17, no. 5, pp. 454-466, May 1991

[31] D. P. Freedman and G. M. Weinberg, *Handbook of Walkthroughs, Inspections, and Technical Reviews: Evaluation Programs, Projects, and Products*, 3rd ed., Little, Brown and Company, 1982

[32] K Futatsugi, J. A. Goguen, J. P. Jouannaud, and J. Meseguer, "Principles of OBJ2," *Proc. 12th ACM Symp. on Principles of Programming Languages*, New Orleans, pp. 52-66, 1985

[33] C. Gane and T. Sarson, *Structured System Analysis: Tools and Techniques*, Prentice-Hall, 1979

[34] C. Gane, *Computer-Aided Software Engineering: The Methodologies, the Products, and the Future*, Prentice-Hall, 1990

[35] M. Gogolla, *An Extended Entity-Relationship Model*, Lecture Notes in Computer Science 767, Springer-Verlag, 1994

[36] I. J. Hayes, "Specification Directed Module Testing, " *IEEE Transactions on Software Engineering*, vol. SE-42, no. 1, January 1986

[37] B. Hetzel, *The Complete Guide to Software Testing*, 2nd ed., A Wiley-QED Publication, 1988

[38] C. A. R. Hoare, *Communicating Sequential Processes*, Prentice-Hall, 1985

[39] M. Holcombe, "An Integrated methodology for the specification, verification and Testing of Systems," *Proc. 1st European International Conference on Software Testing Analysis & Review*, London, October 1993

[40] H. M. Horcher, "The Role of Formal Specifications in Software Test," *Proc. 2nd European International Conference on Software Testing Analysis & Review*, Brussels, October 1994

[41] L. Ingevaldsson, *JSP - A Practical Method of Program Design*, Studentlitteratur and Chartwell-Brett Ltd., 1986

[42] Institute of Electrical and Electronics Engineers, *IEEE Standard Glossary of Software Engineering Terminology*, ANSI/IEEE Std 729-1983, 1983

[43] Institute of Electrical and Electronics Engineers, *IEEE Standard for Software Test Documentation*, ANSI/IEEE Std 829-1983, 1983

[44] Institute of Electrical and Electronics Engineers, *IEEE Guide to Software Requirements Specifications*, ANSI/IEEE Std 830-1984, 1984

[45] Institute of Electrical and Electronics Engineers, *IEEE Standard for Software Verification and Validation Plans*, ANSI/IEEE Std 1012-1986, 1986

[46] Institute of Electrical and Electronics Engineers, *IEEE Recommended Practice for Software Design Descriptions*, IEEE Std 1016-1987, 1987

[47] P. Jalote, "Testing the Completeness of Specifications," *IEEE Transactions on Software Engineering*, vol. 15, no. 5, pp. 526-531, May 1989

[48] C. B. Jones, *Software Development: A Rigorous Approach*, Prentice-Hall, 1980

[49] C. B. Jones, *Systematic Software Development Using VDM*, Prentice-Hall, 1986

[50] C. Kaner, J. Falk, and H. Q. Nguyen, *Testing Computer Software*, 2nd ed., Van Nostrand Reinhold, 1993

[51] R. A. Kemmerer, "Testing Formal Specifications to Detect Design Errors," *IEEE Transactions on Software Engineering*, vol. SE-11, no. 1, pp. 32-43, January 1985

[52] N. L. Kerth, "The Use of Multiple Specification Methodologies on a Single System," *Proc. 4th International Workshop on Software Specification and Design*, pp. 183-189, CS Press, 1987

[53] C. H. Kung, "Conceptual Modeling in the Context of Software Development," *IEEE Transactions on Software Engineering*, vol. 15, no. 10, pp. 1176-1187, October 1989

[54] S. Lee and S. Sluizer, "SXL: An Executable Specification Language," *Proc. 4th International Workshop on Software Specification and Design*, pp. 231-235, CS Press, 1987

[55] B. P. Leintz and E. B. Swanson, *Software Maintenance Management*, Addison-Wesley, 1980

[56] N. G. Leveson, "Guest Editor's Introduction Formal Methods in Software Engineering," *IEEE Transactions on Software Engineering*, vol. 16, no. 9, September 1990

[57] N. Levy, A. Piganiol, and J. Souquieres, "Specifying with SACSO," *Proc. 4th International Workshop on Software Specification and Design*, pp. 236-241, CS Press, 1987

[58] D. Lightfoot, *Formal Specification Using Z*, Macmillan, 1991

[59] M. D. Lubars, "Schematic Techniques for High Level Support of Software Specification and Design," *Proc. 4th International Workshop on Software Specification and Design*, pp. 68-75, CS Press, 1987

[60] J. Martin, *Computer Database Organization*, Prentice-Hall, 1982

[61] J. Martin, *System Design From Provably Correct Constructs*, Prentice-Hall, 1985

[62] J. D. Moffett and M. S. Sloman, "A Case Study in Representing a Model: to Z or not to Z," *Proc. 5th Annual Z User Meeting*, Oxford, December 1990, Springer-Verlag, 1991

[63] S. L. Montgomery, *Relational Database Design and Implementation Using DB2*, Van Nostrand Reinhold, 1990

[64] B. Moret, "Decision Trees and Diagrams," *ACM Computing Survey*, vol. 14, no. 4, pp. 593-623, September 1977

[65] G. J. Myers, *The Art of Software Testing*, John Wiley & Sons, 1979

[66] Takeshi Nakajo and Hitoshi Kume, "A Case History Analysis of Software Error Cause-Effect Relationships," *IEEE Transactions on Software Engineering*, vol. 17, no. 8, pp. 830-838, August 1991

[67] D. Neilson, "Machine Support for Z: The zedB Tool," *Proc. 5th Annual Z User Meeting*, Oxford, December 1990, Springer-Verlag, 1991

[68] M. A. Ould and C. Unwin (Eds.), *Testing in Software Development*, Cambridge University Press, 1989

[69] L. J. Peters and L. L. Tripp, "Comparing Software Design Methodologies," *Datamation*, pp. 316-321, November 1977

[70] J. Peterson, "Petri Nets," *ACM Computing Survey*, vol. 9, no. 3, pp. 223-252, September 1977

[71] F. Polack, M. Whiston, and P. Hitchcock, "Structured Analysis - A Draft Method for Writing Z Specifications," *Proc. 6th Annual Z User Meeting*, Cambridge, December 1994, Springer-Verlag, 1992

[72] F. Polack, M. Whiston, and K. C. Mander, "The SAZ Project: Integrating SSADM and Z," *Proc. 1st International Symposium of Formal Methods Europe*, Denmark, April 1993, Lecture Notes in Computer Science 670, Springer Verlag, 1993

[73] F. Polack and K. C. Mander, "Software Quality Assurance Using the SAZ Method," *Proc. 8th Annual Z User Meeting*, Cambridge, June 1994, Springer-Verlag, 1994

[74] B. Potter, J. Sinclair, and D. Till, *An Introduction to Formal Specification and Z*, Prentice-Hall, 1991

[75] C. V. Ramamoorthy, A. Prakash, W. Tsai, and Y. Usuda, "Software Engineering: Problems and Perspectives," *Computer*, vol. 17, no. 10, pp. 191-209, October 1984

[76] G. Randell, "Data Flow Diagram and Z," *Proc. 5th Annual Z User Meeting*, Oxford, December 1990, pp. 216-227, Springer-Verlag, 1991

[77] C. Rich, R. C. Waters, and H. B. Reubenstein, "Toward a Requirements Apprentice," *Proc. 4th International Workshop on Software Specification and Design*, pp. 79-86, CS Press, 1987

[78] M. Roper and A. Rahim, "Software Testing Using Design-Based Techniques," *Proc. 1st European International Conference on Software Testing Analysis & Review*, London, October 1993

[79] R. G. Ross, *Entity Modeling: Techniques and Application*, Database Research Group, 1987

[80] P. Rudnicki, "What Should be Proved and Tested Symbolically in Formal Specifications," *Proc. 4th International Workshop on Software Specification and Design*, pp. 190-195, CS Press, 1987

[81] M. Rueher, "From Specification to Design: An Approach Based on Rapid Prototyping," *Proc. 4th International Workshop on Software Specification and Design*, pp. 126-133, CS Press, 1987

[82] L. Semmens and P. Allen, "Using Yourdon and Z: an Approach to Formal Specification," Proc. 5th Annual Z User Meeting, Oxford, December 1990, pp. 228-253, Springer-Verlag, 1991

[83] J. A. Senn, *Analysis & Design of Information System*, McGraw-Hill, 1989

[84] I. Shemer, "Systems Analysis: A Systemic Analysis of a Conceptual Model," *Computing Practices*, vol. 30, pp. 506-512, June 1987

[85] I. Sommerville, *Software Engineering*, Addison-Wesley, 1989

[86] J. M. Spivey, *The Z Notation: A Reference Manual*, Prentice-Hall, 1992

[87] J. F. Stay, "HIPO and Integrated Program Design," *IBM System Journal*, 1976

[88] P. A. Swatman, D. Fowler, and C. Y. M. Gan, "Extending the Useful Application Domain for Formal Methods," *Proc. 6th Annual Z User Meeting, York*, December 1991, Springer-Verlag, 1992

[89] W. Swatout and R. Balzer, "On the Inevitable Intertwining of Specification and Implementation," *Communications of the ACM*, vol. 25, no. 7, pp. 438-440, July 1982

[90] R. B. Terwilliger and R. H. Campbell, "PLEASE: A Language for Incremental Software Development," *Proc. 4th International Workshop on Software Specification and Design*, pp. 249-256, CS Press, 1987

[91] T. H. Tse, *A Unifying Framework for Structured Analysis and Design Models*, Cambridge University Press, 1991

[92] P. T. Ward and S. J. Mellor, *Structure Development for Real-Time System (Volume 1: Introduction & Tools)*, Yourdon Press, 1985

[93] A. I. Wasserman and S. Gutz, "The Future of Programming," *Communications of the ACM*, vol. 25, no. 3, pp. 196-206, March 1982

[94] G. M Weinberg and D. P. Freeman, "Reviews, Walkthroughs, and Inspections," *IEEE Transactions on Software Engineering*, vol. SE-10, no. 1, January 1984

[95] J. M. Wing, "A Larch Specification of the Library System," *Proc. 4th International Workshop on Software Specification and Design*, pp. 34-41, CS Press, 1987

[96] J. M. Wing, "A Study of 12 Specifications of the Library Problem," *IEEE Software*, pp. 66-76, 1988

[97] J. M. Wing, "A Specifier's Introduction to Formal Methods," vol. 23, no. 9, pp. 8-24, *Computer*, September 1990

[98] J. Woodcock and M. Loomes, *Software Engineering Mathematics*, Pitman, 1989

[99] S. S. Yau and J. J. -P. Tsai, "A Survey of Software Design Techniques," *IEEE Transactions on Software Engineering*, vol. SE-12, no. 6, pp. 713-721, June 1986

[100] E. Yourdon and L. L. Constantine, *Structured Design*, Yourdon Press, 1979

[101] E. Yourdon, *Modern Structured Analysis*, Prentice-Hall, 1989

[102] K. Yue, "What Does It Mean To Say That A Specification is Complete," *Proc. 4th International Workshop on Software Specification and Design*, pp. 42-49, CS Press, 1987