TOWARD THE FORMALISATION OF USE CASE MAPS

by

CYRILLE DONGMO

submitted in accordance with the requirements

for the degree of

MASTER OF SCIENCE

in the subject

COMPUTER SCIENCE

at the

UNIVERSITY OF SOUTH AFRICA

SUPERVISOR: PROF J.A VAN DER POLL

November 2011

Abstract

Formal specification of software systems has been very promising. Critics against the end results of formal methods, that is, producing quality software products, is certainly rare. Instead, reasons have been formulated to justify why the adoption of the technique in industry remains limited. Some of the reasons are:

- Steap learning curve; formal techniques are said to be hard to use.
- Lack of a step-by-step construction mechanism and poor guidance.
- Difficulty to integrate the technique into the existing software processes.

Z is, arguably, one of the successful formal specification techniques that was extended to Object-Z to accommodate object-orientation. The Z notation is based on first-order logic and a strongly typed fragment of Zermelo-Fraenkel set theory. Some attempts have been made to couple Z with semi-formal notations such as UML. However, the case of coupling Object-Z (and also Z) and the Use Case Maps (UCMs) notation is still to be explored.

A Use Case Map (UCM) is a scenario-based visual notation facilitating the requirements definition of complex systems. A UCM may be generated either from a set of informal requirements, or from use cases normally expressed in natural language. UCMs have the potential to bring more clarity into the functional description of a system. It may furthermore eliminate possible errors in the user requirements. But UCMs are not suitable to reason formally about system behaviour.

In this dissertation, we aim to demonstrate that a UCM can be transformed into Z and Object-Z, by providing a transformation framework. Through a case study, the impact of using UCM as an intermediate step in the process of producing a Z and Object-Z specification is explored. The aim is to improve on the constructivity of Z and Object-Z, provide more guidance, and address the issue of integrating them into the existing Software Requirements engineering process.

Keywords: Semi-formal specification techniques, UCMs, Formal methods, Z, Object-Z, Software Process, Specification Validation, Comparing Specifications, Spiral Model.

Contents

List of Tables

List of Figures

List of publications

1. Use Case Maps as an Aid in the Construction of Formal Specification, MSVVEIS, In the Proceeding of the 7th International Workshop on Modeling, Simulation, Verification and Validation of Enterprise Information Systems (2009).

2. A Four-Way Framework for Validating a Specification, in the Proceedings of the SAIC-SIT'10 conference, ACM (2010).

3. Evaluating Software Specifications by Comparison, in the Proceedings of the SAIC-SIT'11 conference, ACM (2011).

Important Note: Please observe that due to the (automatic) carriage return by Latex, the word **SAICSIT**, in the second and third publications above, is presented as **SAIC - SIT**. A number of such cases are encountered in other parts of this dissertation, I couldn't find a way to fix it. I hereby apologise for any inconvenience on the presentation of this work.

Acknowledgements

In the first place, I would like to express my sincere gratitude to my supervisor, Professor John Andrew van der Poll, whose advice and guidance justify the achievements of this dissertation. Without his help, it would have been hard to bring this work to its present state.

I would like to convey my gratitude to Professor Amyot (from the School of Information Technology and Engineering at the University of Ottawa, Canada) and his team for making most of the publications on UCM available on their website and for providing the two UCM tools: UCMNav and jUCMNav. Without these, it would have been very hard to draw all the UCMs diagrams in this dissertation.

I also extend my thanks to the Research Directorate at Unisa (University of South Africa) firstly, for organising training workshops through which I gained very helpful insights on how to conduct a research and report the results. And secondly for the grant through the Masters and Doctoral Support Programme (MDSP). In the same vein, I wish to send a huge thanks to the College Research and Ethics Committee (CREC) for funding all my participation to the national and international conferences where the main contributions of this work were presented, as full research papers, and discussed with peers.

Finally, I would like to express my deepest recognition to the people closest to me. To my dearest wife Solange, thank you for your love, support, and understanding. Thank you to my 3 year old friend and son Lemek, my 1 year old daughter Loriane, and to all my brothers and sisters.

C. Dongmo
November 2011

Dedication

This dissertation is dedicated to the memory of my father, Albert Dongmo (1924-1998), to the memory of my very dear friend and elder sister, Rachelle Dongmo (1967 - July 2011), to my mother, to my wife Solange, to my son Lemek (aged 3 years) and to my daughter Loriane (aged 1 year).

Chapter 1

Introduction

1.1 Context and motivation

Specifying a software system formally, implies in general, the use of a specification language based on mathematics (e.g. set theory and predicate logic) to describe precisely the properties of the system (Henderson [39], O'Regan [66]). Formal methods came into software engineering with a great deal of promise. Arguably, it may be used in every phase of software development life cycle (SDLC) to produce quality products, provides detailed and correct requirement specifications, and detects ambiguous, incomplete and inconsistent statements in system requirements at an early stage of the system development process. A formal requirements specification also offers the advantage being potentially amenable to automated reasoning and analysis (Nuseibeh and Easterbrook [65], van der Poll [90]).

However, after a long period of intensive research and development, the use of "formal methods" in industry is still limited (see Abrial [2], Knight et al. [49]). A number of possible reasons have been raised: amongst others, high initial cost (arguable, because the incurred cost at the specification phase, is said to be compensated for at the later design and implementation stages), and a steep learning curve, due to the limited mathematical skills of software engineers and practitioners resulting in a limited number of Formal methods experts. Conversely as Bowen and Hinchey [14] suggest, "*Thou shalt have a Formal method guru on call*". However, van Lamsweerde [95] observes that formal specifications are hard to develop and assess because of the diversity and subtlety of errors that can be made, and the multiplicity of modelling choices that can be considered. Similarly, formal techniques (together with their resulting products), are said to be isolated from other software products, and processes both vertically and horizontally:

1. ***Vertical isolation*** describes a twofold gap; the first between the initial goals, require-

ments, domain constraints, etc. and the resulting formal specification. The second, between the formal specification and the high level-design, leading to the final product. Figure 1.1 illustrates these gaps, and raises two important questions to address the issue about the integration of formal specifications into the entire software development process. Firstly, the need to investigate, how initial goals are refined, user requirements are captured, defined and analysed using formal methods. Secondly, the techniques, processes or tools allowed, and how they are related to final specifications. The next question involves the refinement of formal specifications, into final products.

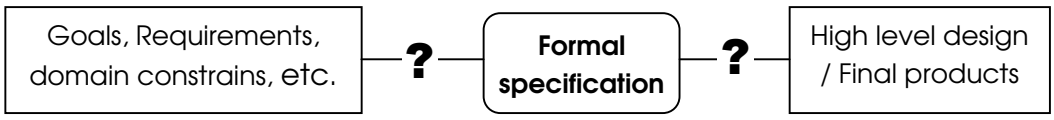


Figure 1.1: Illustration of vertical isolation

2. ***Horizontal isolation*** is specifically concerned with other software products formal specifications should be linked to.

It is also suggested that formal specification techniques do not provide enough guidance for a system specification (van Lamsweerde [95]). This implies the limited, or non-existence, of a systematic constructive mechanism for building complex specifications in a step-by-step approach. This raises the same problem mentioned above, about the ***vertical isolation*** of formal specifications, that is, the need to investigate the process of constructing a formal specification from scratch (requiring initial goals or requirements from users). To this end, two alternative views are plausible: the first, is to consider a formal specification approach as a complete process. The second, is to regard it as a step within the software specification process, that needs to be linked to other existing methods.

The idea of considering a formal specification approach as a complete specification process is largely rejected by the literature typified by this quote from Bowen and Hinchey [14] "*Thou shalt not abandon thy traditional development methods*". A possible reason, is that the main characteristics of such approaches stem from the specification notation languages used at a specific phase of the design process: *Thou shalt choose an appropriate notation.* Examples of formal specification languages (methods) are: VDM, B, Z , Z++, Object-Z, etc. It may be important to observe, that formal techniques also follow (like other specification techniques and methods) the generic Requirements Engineering process, that broadly includes two inter-related phases: a phase during which requirements are elicited and analysed, and the specification and validation phase. As illustrated in Figure 1.2, it is strongly suggested, that formal techniques be introduced into the process, at the specification and

validation phases.

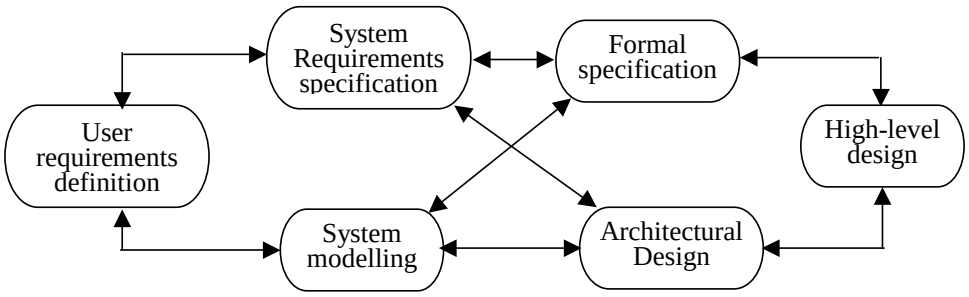


Figure 1.2: Formal specification in Software Development Process (Sommerville [82])

Thus, research has been directed at finding a way to couple existing semi-formal techniques, (e.g. Use Cases, Unified Modelling Language (UML), Use Case Maps (UCM), etc.), that are said to be more suitable at the initial phase, with formal specification methods. Amongst others, the following cases from the literature are illustrative: Coupling UML and B as suggested by Snook and Butler [81], that resulted in creating a new method, namely U2B, UML and TROLL (UML-TROLL) by Gogolla and Richters [33], generating Object-Z specifications from Use Cases (advocated by Moreira and Araújo [63]), and translating UCM diagrams to Communicating State Machine specifications(UCM-ROOM design method) put forward by Bordeleau and Buhr [12]. Other similar cases are found in: Ledru [51], Matta et al. [55], Wieringa et al. [97].

A Use Case Map is a scenario-based, semi-formal specification technique, that gained popularity due to its applicability and adaptability for various purposes. A UCM model may be generated from a set of informal requirements, or use cases, expressed in natural language. It facilitates the understanding, by humans, of large and complex systems by combining, in a single view, the behavioural and architectural structure of the system. The notation also has the advantage of facilitating the capture and definition of requirements during the needs analysis phase. As mentioned above, some attempts have been made to translate UCMs into other languages, including a number of formal notations. However, little is known about coupling UCM with Z and Object-Z. The absence of transformations from UCM to Z and Object-Z (apart form this work) is confirmed in a systematic literature survey on URN Amyot and Mussbacher [8].

1.2 Problem statement

As mentioned above, the constructivity problem: the lack of a step-by-step methodology and poor guidance in the construction of formal specifications, are amongst the key limitations of formal specification techniques. This dissertation intends to investigate the impact (on the final Object-Z specification), of using the semi-formal method UCM in the process of constructing a Z and an Object-Z specification. To this end, the research questions are:

RQ 1: Are UCM models ***transformable*** to Z and Object-Z specifications? In other words, can UCM models of a system, be used as inputs for generating Z and Object-Z specifications?

RQ 2: What would the impact of UCMs on be the ***quality*** of a Z and an Object-Z specification obtained by transforming a UCM model?

RQ 3: What would the impact of UCMs be on the ***process*** of constructing Z and Object-Z specifications, if the ***specification process*** starts with UCM?

The reasons for addressing these questions are presented next.

1.2.1 Research objectives

This work aims to achieve the following:

- Demonstrate that the use of a UCM method can complement Z and Object-Z, by providing a mechanism to transform a UCM model of a system, into Z and Object-Z specifications.
- Demonstrate the usefulness of UCM in the process of constructing Z and Object-Z specifications, by evaluating the **quality** of an Object-Z specification obtained from a UCMs model.
- Improve the constructivity of Z and Object-Z, by suggesting a **step-by-step methodology**, whereby a UCM is used as an intermediate step.

The use of the UCM modelling technique in the construction of Z and Object-Z specifications can, it is argued, significantly improve the quality of specification and provide more guidance.

1.3 The research approach

To address the above problem, a threefold research approach is adopted, involving a case study approach, content analysis and comparative approaches (Hofstee [40]). For the case

study, a requirements definition is given in a natural language (English). Two Object-Z specifications are derived from it following two different construction paths, as illustrated in Figure 1.3. Since the object-oriented extension of Z, Object-Z mostly affects the structuring

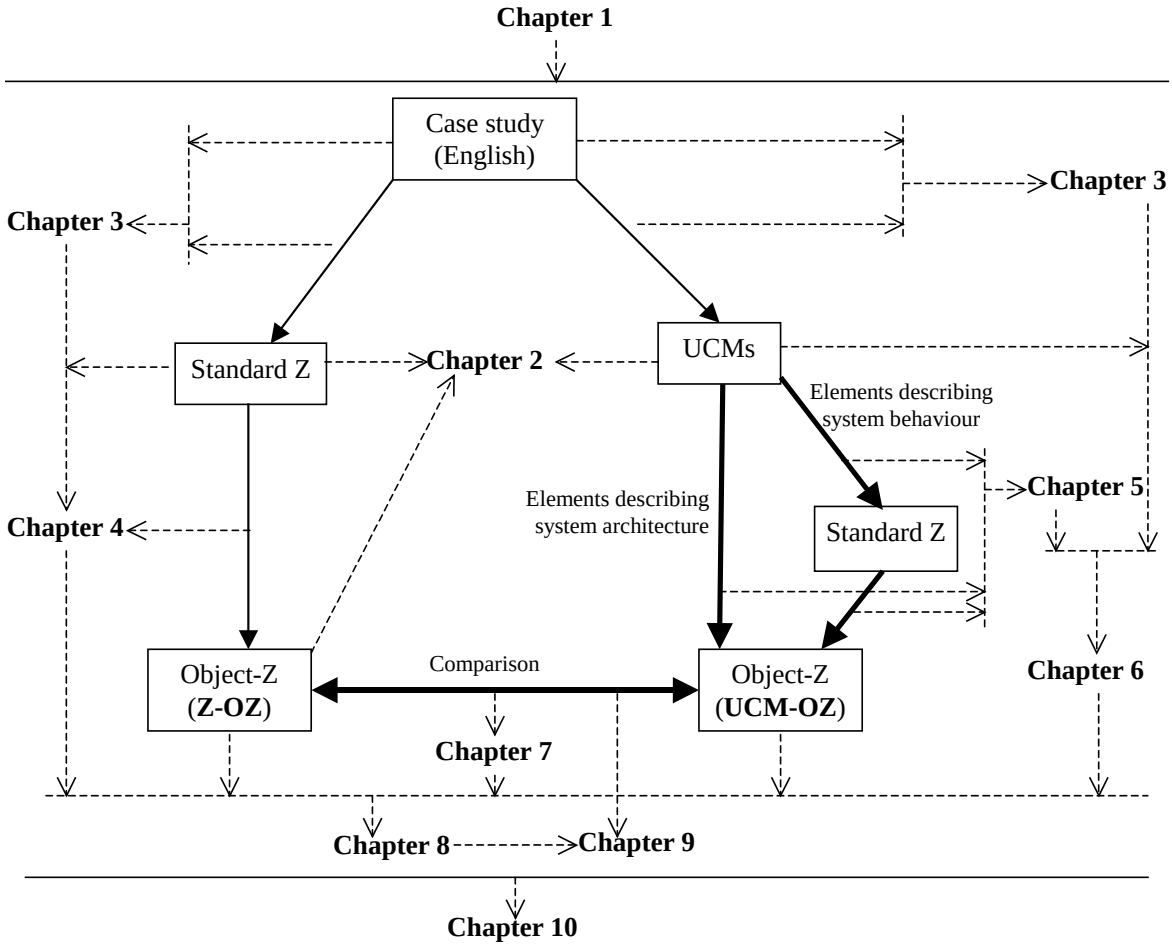


Figure 1.3: Research strategy

of Z components, but makes very little changes to the Z description of the functionalities of a system (content of Z components). Z is maintained as an intermediate step in each of the two specification processes adopted in this dissertation.

Path1: **(Informal requirements → Z → Object-Z)**. To validate the suggested process in Figure 1.2, this path was deliberately chosen to observe the impact of moving directly from an informal set of requirements, to a formal specification. Therefore, the initial requirements are informally analysed so as to identify objects and operations that are described in Z. Then, the Z specification is translated into Object-Z, i.e. the Z-OZ specification.

Path2: (**Informal requirements → UCMs → (Z specification → Object-Z)**). The initial requirements are first translated into UCMs models. Since UCMs allow a

single model to encapsulate both the architectural structuring of components, and the description of the functionalities of the system, this approach proposes translating the structure of a system into meta-data of Object-Z, and then translating the functionalities to Z then, to Object-Z, to complement the meta-data. This process produces an Object-Z specification, says UCM-OZ .

Each of the resulting specifications is evaluated separately (content analysis approach) to identify its quality. Based on the validation results, the two specifications are compared (comparative analysis approach), to measure the impact of using UCMs in the process. The transformability of UCMs to Z and Object-Z is investigated in Path2 when moving from a UCM, to Z and Object-Z. The thick and bold arrows indicate the area from which the main contributions were derived. Dashed arrows indicate the chapters of the dissertation, and the links between them.

1.3.1 Weakness of the approach

A possible limitation of the approach is the risk of subjectivity. Since the same person designs both specifications, it is plausible that insights gained from constructing one specification, may influence the other. However, the effect would be limited, since a specification is not done continuously from the beginning to the end. Intermediate actions, involving different activities, are performed between the specification phases.

1.4 Significance of the research

This work is an intermediate phase towards exploiting the benefits of UCMs, to ameliorate the complexities of formal specification techniques. The aim is to create an iterative and interactive environment for generating Z and Object-Z specifications, where UCMs serve as an interface. The work also aims to provide a mechanism to evaluate and compare software specifications.

1.5 Dissertation layout

As mentioned above, chapters and the links between them are depicted in Figure 1.3. Chapter 2 presents an overview of the literature on UCMs, Z and Object-Z. It starts with an overview of UCMs, where the general concepts of the notation and elements of UCMs are presented. This is followed by a summary of some extensions of the original notation, proposed in the literature and the available tool support. Thereafter, an overview of Z and

Object-Z are presented. Important concepts are illustrated, with examples, as well as a list of a number of Z and Object-Z tool supports.

Chapter 3 focuses on the case study. It defines user requirements and indicates the approach to be followed. A UCM model, as well as a Z specification, are derived from the case study. Some observations made during the UCM modelling and the Z specification are noted.

Chapter 4 aims to transform the Z specification of the case study, into Object-Z. The transformation process is first presented, followed by the Object-Z transformation of the input Z document, where Object-Z class schemas are created to encapsulate Z elements.

Chapter 5 proposes a framework mechanism, to generate Z and Object-Z specifications from a UCM. The basic transformation strategy is first presented, followed by an analysis of the conceptual relationship between the three specifications UCM, Z and Object-Z. Concepts in these three notations are analysed, and a set of guidelines is proposed for the transformation process.

Chapter 6 applies the framework, proposed in Chapter 5, to the UCMs of the case study to generate a Z and an Object-Z (UCM-OZ) specification. A UCM stubbing technique is first applied to the input UCM, to split it into sub-maps, as recommended by the framework. Then, each sub-map is transformed individually. Where necessary, formulas are included in the class schemas to describe the relationships between resulting sub-systems such, as inheritance, polymorphism, etc.

Chapter 7 proposes a generic framework to guide the validation of a software specification. A conceptual relationship between a software specification and four aspects of a system is analysed. These are stakeholder expectations; the application domain; notation language and tool support; and finally, the envisioned software product. The characteristics of a quality software specification are briefly explored, followed by the proposed specification validation strategy, based on Boehm's spiral model.

Chapter 8 applies the framework in Chapter 7, to Z-OZ and UCM-OZ. A common scope for the validation is first defined. A sample list of properties expected from a satisfactory specification is identified and related to stakeholder expectations, by means of mathematical formulas. This is followed by a brief presentation of the validation criteria. The two specifications, Z-OZ and UCM-OZ, are respectively validated relative to each property identified earlier.

Chapter 9 presents an analysis of the results of the validation in Chapter 8, by comparing Z-OZ and UCM-OZ. The aim is to evaluate the impact of using UCMs in the process of generating Z and Object-Z documents. The analysis approach is first presented where guidelines are defined. This is followed by a comparison of the two specifications relative to the list of properties identified earlier in Chapter 8. The result of the comparison is shown

in a tabular form.

Chapter 10 concludes this dissertation. It summarises the main findings and relates them to the research questions, as well as presenting the advantages of this work and highlights further research which could be undertaken.

Chapter 2

Introduction to Use Case Maps, Z and Object-Z

This chapter introduces the three modeling and specification languages used in this dissertation: the Use Case Maps (UCMs), the Z and Object-Z notations. Since each of these notations covers a large spectrum of concepts, for space purpose, only those concepts that are used in this dissertation are discussed, starting with UCMs, then followed by Z and Object-Z.

2.1 Use Case Maps

The UCMs modelling technique was proposed by Buhr and Casselman [20] to document and view a system (that Buhr called behavioural fabric) as mentally perceived by a designer in the light of requirements. Such perception has not been documented in software development process, the model thus, aims to bridge the gap between User requirements and design models (see Buhr [17, 19]). Figure 2.1 illustrates a UCM model documenting high level design constructs to represent a designer's perception of how the behaviour of a system forms part of in the development process.

Use Case Maps accept as inputs user requirements, either expressed in natural language, or transformed into Use Cases (as indicated by Amyot [4]). A Use Case may be described as a set of scenarios, which are sequences of actions performed by the system to yield an observable result to its environment (see Booch et al. [11]). A Use Case Map as a scenario-based notation, describes in an abstract way, how the organisational structure and the emergent behaviour of a system are intertwined (van der Poll et al. [94]). It gives a road-map-like view of the cause-effect paths, traced through a system by scenarios, in a compact map. It enables many scenario paths to be expressed in a single diagram in a way that reveals patterns and

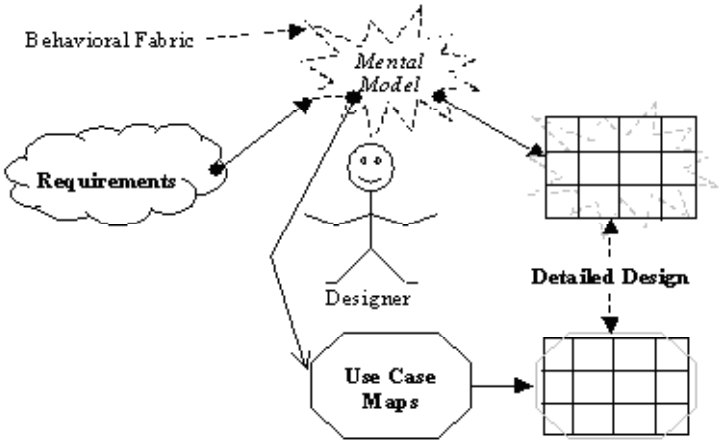


Figure 2.1: Documenting a behavioural fabric

saves them for reuse (see Buhr [17]). Figure 2.2 shows a simple UCM model that illustrates the graphical representation of some UCM elements. A Use Case Map basically comprises

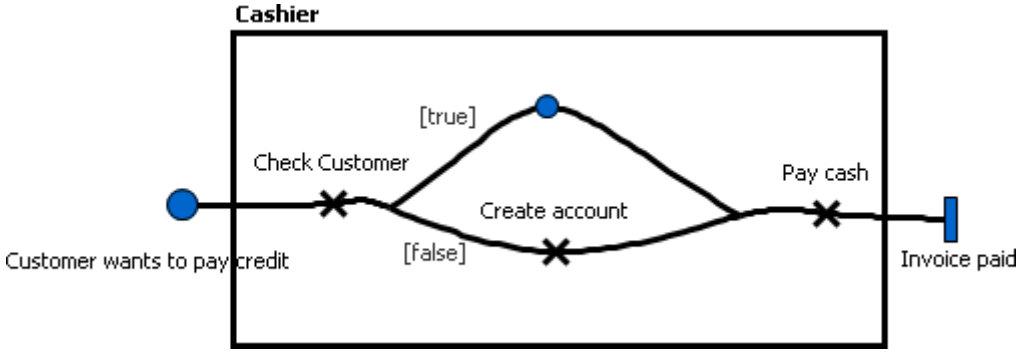


Figure 2.2: An example of a UCM model

a set of abstract components, discussed below, to describe the organisational structure of a system and a set of paths to describe Use Cases.

The UCM elements that appear on Figure 2.2 are explained in the next Sections.

2.1.1 UCM abstract components

An abstract component may be viewed as a self-contained operational unit with internal state and links that enable the component to interact with others. Each component is responsible for performing responsibility points located in it and chained with path segments. Different types of components are provided by the UCM notation: Team, Process and Object (see Figure 2.3).

Figure 2.3: Abstract components

Team

A Team component is a generic component allowed to contain any other component type including other teams. It is represented graphically by a rectangle.

Process

A Process is an autonomous, active component, that may operate concurrently with other processes. A process may contain passive components, those that do not have control over the responsibilities that they perform, such as Objects. It is represented graphically by a parallelogram.

Object

An Object is a passive component, that supports data or procedural abstraction through an interface. Objects perform their own responsibilities but do not have ultimate control of when they are activated.

2.1.2 Basic path notation

Figure 2.4 shows an example of basic path notation. It comprises: a Start Point, a Path Segment, a Responsibility Point, and an End Point.

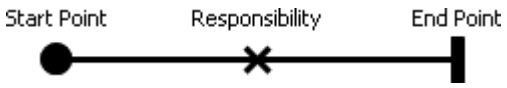


Figure 2.4: Basic UCM path notation

Start point

A Start Point is represented graphically by a filled circle. It is defined as a set of possible triggering events and optionally a precondition. The execution of a path begins when some triggering events occur with the precondition enabled.

Responsibility

A Responsibility Point is represented graphically by a cross. It illustrates a generic processing that is to be performed, which can be for example, an operation, a task, an action, a function and so forth.

End Point

An End Point is represented by a vertical bar and is defined by a set of resulting events and an optional post-condition that terminates the execution of a path.

Path Segment

A UCM path segment is represented graphically by a continuous line with any possible and unambiguous shape. It may sometimes be useful to indicate the direction of a path segment, but in general it is not necessary. A path segment is used to express an ordered sequence of UCM elements that require to be executed.

Use Case Maps provide the concept of path connectors to describe alternative use cases, and parallel executions of scenarios.

2.1.3 Path connectors

A UCM path is the execution route of one or more scenarios, and may be composed of a number of path segments, interconnected by means of path connectors to achieve path coupling, and express interactions between scenarios. Amongst others, path connectors are: OR-forks, OR-joins, AND-forks, and AND-join (see Figure 2.5).

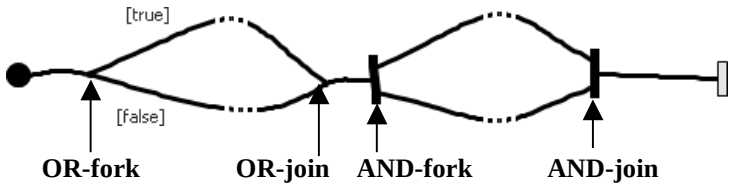


Figure 2.5: Path connectors

OR-forks

An OR-fork splits a path segment into two or more branches. Alternative path segments may be guarded by conditions, depicted inside square brackets. For example, in Figure 2.2, the condition [false] indicates that the customer wanting to make a payment does not have a valid account in the system.

OR-join

An OR-join is a place on a UCM diagram where two or more path segments merge into a single one. The merging of the path segments does not require any synchronisation or interaction between the incoming paths.

AND-forks

An AND-fork is represented graphically by a vertical ticked bar that splits an incoming path segment into two or more parallel paths. This connector helps to represent the concurrent progression of scenarios along path segments.

AND-join

An AND-join connector collapses two or more parallel paths into a single one. It is represented graphically by a vertical bar.

The AND-fork/join elements provide a strong form of representing inter-scenario synchronisation in which scenarios along different paths are mutually synchronised. The OR-fork/join UCM concept allows for multiple scenarios to progress along a single path segment and be separated independently only where necessary.

Two types of path elements called **stubs** are discussed next.

2.1.4 Stubbing techniques

A UCM provides for the concept of stubs to help sub-divide complex maps into two or more sub-maps. A stub is a mechanism for (paths) abstraction that represents on a UCM diagram, a place where a sub-map is needed, but for which details are referred to elsewhere. It saves as maps connectors that help to link the execution of a scenario from a map containing the stub (called root-map) to a sub-map called a "plug-in". The two types of stubs are: static-stubs, and dynamic-stubs (see Figure 2.6).

Static-stub

When only a single sub-map is needed, a static-stub is used. The binding of the plug-in to the root-map is made as follows: the input path segment(s) entering the stub (generally noted INX, where X stands for a referencing number) is (are) associated to the Start point(s) of the plug-in, and the End point(s) of the plug-in is (are) associated to the output path segment(s), leaving the stub (generally noted by $OUTX$ as in Figure 2.6). This association

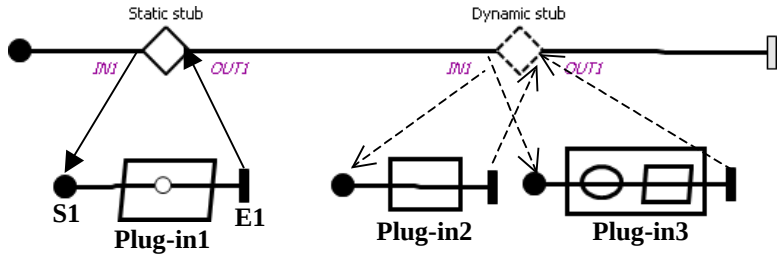


Figure 2.6: An example of a static and a dynamic stub

is called a *Binding Relationship*. In the case of the static-stub in Figure 2.6, it is indicated by:$\{\langle IN1, S1\rangle, \langle OUT1, E1\rangle\}$ (Amyot [6]).

Dynamic-stub

A dynamic-stub is used where more than one alternative sub-diagram is needed, for which the binding to a specific diagram is determined during the execution of the scenario being modelled. A selection policy to determine the plug-in to execute is, therefore defined.

Other key notation elements are: Failure-point, Waiting-place and Timer. These are presented next, through the Timeout-recovery mechanism that is provided - by UCM - to model the enhancing of network failures in a network communication.

2.1.5 Timeout-recovery mechanism

Figure 2.7 shows the graphical representation of the three UCM elements: Failure-point, Waiting-place and Timer; it also includes a model for a Timeout-recovery mechanism. In

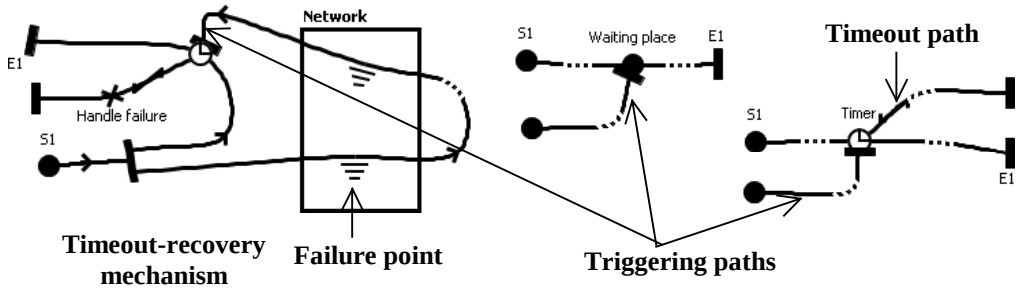


Figure 2.7: An application of the use of a UCM Failure-point and Waiting place

each of the three components in the figure, the path from start-point $S1$ to the end-point $E1$, is called the *main path*. It is the path on which a scenario progresses to reach the waiting-place or timer. The triggering paths are also indicated. Those are paths along

which triggering events occur, to cause a waiting scenario to continue progressing along the main path.

Failure-point

A failure-point indicates a place along a path where the progression of a scenario may stop leaving the system in an incomplete state, possibly jeopardising other paths in execution. For example, a network communication may fail, causing a sent message not to reach its destination or an acknowledgement not to reach the sender.

Waiting-place

A waiting-place indicates a place where a scenario progressing along a main path, may need to pause waiting for an event to occur along the triggering path, before it continues. The triggering path may terminate at the waiting-place or touch it tangentially, and continues. Identifying a path as a main path, or triggering path, is relative, since the same path may play both roles depending on the scenario under consideration.

Timer

A Timer, also known as a timed waiting-place, is just a variation of a waiting-place that uses a time clock to control the occurrence of the triggering event. The timeout path on the diagram in Figure 2.7 is used to model the situation when the waiting time expires before the occurrence of the triggering event.

With the Timeout-Recovery mechanism in Figure 2.7, a message is sent via the network component and concurrently, the Timer is set up to wait for an acknowledgment that may be sent back via the network. If the acknowledgment is not received before timeout, then network communication failure is assumed, and the responsibility point labeled *Handle failure* is performed. Otherwise, the execution continues to the end-point $E1$.

2.1.6 Extending the original UCM notation

As mentioned earlier, a UCM aims to bridge the gap between user requirements and detailed design (Buhr [17, 18], Buhr and Casselman [20]). Its core notation does not completely cover the notational needs in some specific application domains. Some extensions have been proposed, either to the basic features of a UCM, or to its applicability. Some notational elements and concepts were added to the basic UCM features to support the agent systems (Amyot [4]). As reported by van der Poll et al. [94], UCM support for designing user interfaces is still

acknowledged to be insufficient. In this regard, an extension of the basic UCM, that reinforces the exchange of messages between users and the system aimed at allowing the notation to adequately support the user interfaces and usability requirements analysis and modeling was suggested. For a similar reason, a number of heuristics were proposed to facilitate the validation of the three important properties: consistency, completeness and precision.

UCMs efficiently address functional requirements, but leave non-functional requirements uncovered. The visual notation language, GRL (Goal-oriented Requirement Language[1]), is used to describe business goals, non-functional requirements, alternatives, and rationales. Aiming to capitalise on the advantages of each of the two notations, Amyot and Mussbacher [7] proposed combining UCMs and GRL notations, into a single notation, namely, URN (User Requirements Notation), which is now standardised as reported in Amyot [5].

2.1.7 UCM tool support

Two freely available editing tools now support UCMs: the oldest is UCM Navigator (UCM-Nav) (Miga [59]) and more recently jUCMNav (Mussbacher and Amyot [64], Roy et al. [72]).

UCMNav is a graphical software system that helps to create UCMs diagrams. This tool supports most of the features defined in the UCM reference manual (Buhr and Casselman [20]). It maintains binding between plug-ins and stubs, responsibilities to components, sub-components to components etc. It allows users to visit and edit the plug-ins related to stubs at all levels. It loads, exports and imports UCM as XML files. It can also export a UCM diagram to formats such as Encapsulated Postscript (EPS), Maker Interchange Format (MIF), and Computer Graphics Meta-file (CGM). As reported by Kealy [48], the main drawback of this tool is that it is hard to install and maintain.

The jUCMNav tool is a user-friendly graphical editor under the Java-based open-source Eclipse platform. As an improved version of UCMNav, it provides more functionality including a support for Goal-oriented Requirements Language(GRL). Its export-import possibilities are various, and include the generation from an input UCM of different types of files such as XML files, MSC (Message Sequence Charts) files, and the CSM (Core Scenario Model) files.

The following section presents an overview of the Z specification language.

2.2 Z specification

Z (pronounced 'zed') is a formal specification notation based on the first-order predicate logic and a strongly-typed fragment of Zermelo-Fraenkel (ZF) set theory (see e.g. Lightfoot [52], Mole [62], O'Regan [66], Spivey [83]). The notation was initiated by Jean-Raymond Abrial in France, and developed at the Programming Research Group (PRG) of Oxford University, in England, since the late 1970s. The main construct in Z is called schema, which is built upon basic types and global variables.

2.2.1 Basic types and global sets

The concept of a basic type (also called a Given Type), is provided in Z, to specify the set of elementary objects, for which details are left unspecified. The list of basic types, for a specification, is enclosed inside square brackets and separated by commas. For example:

$$[Customer, Book, Account]$$

defines a list of basic types in Z, for which for example, *Customer* specifies the set of all possible customers. Detail information about customers, books and accounts are deferred to the design phase. A basic type may be used anywhere in the specification after its definition (see Bowen [13]).

Similar to basic types, a global variable may be used anywhere in the specification after its definition. The axiomatic definition of a global variable is presented as follows:

$$\begin{array}{|l} \textit{declaration part} \\ \hline \textit{predicate part} \end{array}$$

For example:

$$\begin{array}{|l} max : \mathbb{N} \\ \hline max \leq 50 \end{array}$$

The concept of **Free types** is also used to list, for a type, the identifiers of its element. The general form is:

$$freetype \ ::= \ element_1 \mid element_2 \mid ... \mid element_n$$

E.g. $Response \ ::= \ yes \mid no$
The central concept in Z is the Schema introduced next.

2.2.2 Z schemas

The general form of a schema is:

SchemaName

declarations
predicate

SchemaName represents the name of the schema. The *declarations* include a list of typed variables, called *components*, which are constructed from a list of Basic types identified during the construction of a Z specification. The *predicate* defines constraints or relationships between the components in the declaration part. The abbreviated notation of the above schema is:

$$SchemaName == [declaration\ part \mid predicate\ part]$$

Two types of schemas are encountered: "state schemas", to describe the static behaviour of a system, and "operation schemas" to describe the dynamic behaviour. For illustration purpose, the Airport example below from Lightfoot [52] is considered:

The air-traffic control of an airport keeps a record of the planes waiting to land and the assignment of planes to gates on the ground.

State schema

In Z, an abstract state, also called a state schema, specifies the static behaviour of a system. For example, with the airport example above, assume the given types:

$$[Plane, Gate]$$

Where *Plane* denotes the set of all possible, uniquely identified planes, and *Gate* the set of all gates at the airport. The state schema is:

Airport

$waiting : \mathbb{P}\ Plane$
$assignment : Gate \rightarrowtail Plane$
$waiting \cap (\text{ran}\ assignment) = \varnothing$

The component *waiting* maintains a list of planes waiting to be assigned to a gate, and *assignment* maps each gate to one, and only one, plane. The predicate part indicates that only planes that have not yet been assigned a gate, are kept in the waiting list. An important aspect of a system state is its inherent variability in time, e.g. when a new plane is assigned

a gate, the value of each of the two components *waiting* and *assignment* changes and hence, the state of *Airport*. Z provides an operation called "schema decoration" to describe the change of system states.

Schema decoration

A schema S is decorated by adding a prime to its name (S'). The effect of decorating S, is that all the variables in the declaration and predicate part of S, are also decorated (see Potter et al. [70]). Since an operation performed on a state schema may change the state of the system, an important aspect of schema decoration is to facilitate the specification state change within the operation schema. The state before and after the operation, are both included in the declaration of an operation schema, and related in the predicate part, to show, for example, how state variables are changed by the operation.

Schema as a type

To define composite (complex) structures, Z allows a schema to be used as a type (Jacky [42], van der Poll [90]). Such a type is similar to a record type in conventional programming languages such as Pascal. An instance of a schema type is called a binding. Z provides the unary operator θ to reference each binding. E.g. an instance of the schema *Airport* is:

$$\langle waiting \Rightarrow \varnothing, assignment \Rightarrow \varnothing \rangle$$

For each abstract state space, a realisable initial state is required.

Initialising the state space

It may be assumed that initially, the list of planes in the waiting list is empty and the list of gates assigned to planes is also empty. Therefore the state of the *Airport* is initially represented as:

InitAirport

$$Airport'$$

$$waiting' = \varnothing \land assignment' = \varnothing$$

Although it is relatively easy to observe that this state is realisable, in general, it is recommended to establish that the initial state is realisable. To this end, the initialisation theorem is used:

$$\vdash Airport' \bullet InitAirport$$

This implies the need to demonstrate that there exists a state $Airport'$ of the state space *Airport*, for which the components $waiting = \varnothing$ and $assignment = \varnothing$.

Partial operation

To illustrate the concept of an operation in Z, consider the following schema that assigns a gate to a plane.

assignGate

$$\Delta Airport$$
$$plane? : Plane$$
$$gate? : Gate$$

$$plane? \in waiting$$
$$assignment' = assignment \cup \{gate? \mapsto plane?)\}$$
$$waiting' = waiting \setminus \{plane?\}$$

The delta (Δ) symbol is used to indicate the state schema that the operation changes. The question mark (?) that follows the two variables *plane*? and *gate*? indicates that those are input variables. An exclamation mark (!) is used to denote an output.

The logical expression $plane? \in waiting$ in the predicate part, constraints the input plane to be taken only from the waiting list. This defines the condition under which the operation becomes applicable, i.e the precondition. The precondition of each operation may be calculated (Woodcock [100]) to determine the circumstances under which an operation is applicable. For example, if the input plane is not in the waiting list, an error is generated and further operations are needed to handle the error. Hence, *assignGate* is said to be a partial operation, since further operations may be needed to specify error conditions.

Error condition

As mentioned in the previous section, if a plane used as input in the operation *assignGate* is not in the waiting list, an error occurs and the following operation is specified for the error case.

unknownPlane

$$\Xi Airport$$
$$plane? : Plane$$
$$resp! : Response$$

$$plane? \notin waiting$$
$$resp! = PLANE\_UNKNOWN$$

The symbol Ξ is used to indicate that the operation operates on *Airport* but, does not change its state.

Total operation

In Z, a complete version of the operation that maps each plane to a specific gate may be formed by combining the operation under normal circumstances, and those to handle errors.

$$totalAssignment \mathrel{\widehat{=}} assignGate \lor unknownPlane$$

The definition of the operation *totalAssignment* is a predicate schema expression that uses the Z disjunction operator $\lor$, to combine two operations. The semantics of this operation is the following: The declaration part of the composed operation, is obtained by merging the declarations of each of the individual operations. The predicates of the individual schemas are disjoined. More schema operators are available to facilitate the construction of predicate schema expressions.

Schema calculus

Amongst others, the following operators are provided in Z: schema inclusion, schema conjunction ($\land$), schema negation ($\neg$) and sequential composition ($\fatsemi$), (see e.g. Potter et al. [70]).

(a) **Schema inclusion**: This operator allows the name of a schema $S1$, describing an abstract state space to be included in the declaration part of another state space schema $S2$. The declarations of $S1$ are included in those of $S2$, and the predicate of $S1$ is appended (ored) to that of $S2$.

(b) **Schema negation ($\neg$)**: The negation of a schema S, is a schema denoted by $\neg S$. It has the same declarations as S, and its predicate, is the negation of the predicate of S.

(c) **Schema conjunction ($\land$)**: Let R and S be two schemas, and $P = R \ \land \ S$. P is a schema obtained as follows: the declarations of R and S are merged to form that of P and their predicates are conjoined (anded) to form that of P.

(d) **Schema composition ($\fatsemi$)**: Consider an operation C, defined as: $C = A \fatsemi B$ where A and B are two operation schemas. The semantics of C is the following: if the operation A can change the state of the system from S to $S1$, and B from $S1$ to $S2$, then C is an operation that changes the state of the system from S to $S2$.

Some limitations of Z due to schema calculus and the use of schemas as types were analysed by van der Poll [90]. However, the major disadvantage of using Z for large systems is its inherent lack of object-oriented structures, making it hard to group and manage a rapidly increasing number of schema structures. To this end, the notation was extended to Object-Z to accommode object-orientation (Carrington and Smith [22], Smith [78]). An overview of Object-Z is presented next.

2.3 Object-Z specification

As mentioned by Taylor et al. [86], Object-Z (see Duke and Rose [26], Duke et al. [27], Smith [77]) is one of the most developed of several Z-like Object-Oriented specification languages. It employs the concept of a class schema to encapsulate Z schemas. A class schema is, in general, structured as follows (Smith [77]):

ClassName [*generic parameters*]

[*visibility*]
[*Inherited classes*]
[*local definitions*]
[*state schema*]
[*initial state schema*]
[*operations*]

The *generic parameters* list is optional as is each component of the class. The visibility list denoted by $\restriction$, restricts access to some components and operations of the class. Similarly, the list of inherited classes is optional. Z-like type and constant definitions may be specified. Unlike in Z, operations and the state schemas are described within the class. The order in which those components appear is prescriptive. A class schema may include only one state schema, which is very similar to Z state schemas, and does not carry a name. The components in the state schema may be initialised to some realisable values. The only initial state is named *INIT*. It includes only instances of the components declared in the state schema. Operations are described in the same vein as in Z, with some differences as indicated next.

2.3.1 Operation schema

The concept of an operation in Object-Z is similar to that of Z. The only difference is that an operation in Object-Z operates on a single state schema. The Delta (Δ) operator lists specific components changed by the operation, whereas the Xi (Ξ) operator is simply discarded in Object-Z. The concepts of partial, total operations and error handling, are not provided since an operation in Object-Z becomes applicable only when the precondition of the operation is satisfied. Most of the Z schema calculus operators (e.g. $\lor$, $\land$, $\fatsemi$, *etc.*), are also used in Object-Z. However, the semantics of some of them may vary slightly in the context of a class schema. Additional schema operators are also provided. Two examples are: the nondeterministic choice ([]) and scope enrichment operators ($\bullet$) (see Duke and Rose [26], Smith [77]).

An example of a class schema to specify the airport example from Section 2.2 is given

next. The two components (*waiting* and *assignment*) and the operation *totalAssignment* are made accessible from the system environment.

ClsAirport

$\lceil(waiting, assignment, totalAssignment)$

$[Plane, Gate, Response]$

$waiting : \mathbb{P}\, Plane$
$assignment : Gate \rightarrowtail Plane$

$waiting \cap (\text{ran}\ assignment) = \varnothing$

INIT

$waiting = \varnothing \land assignment = \varnothing$

assignGate

$\Delta(waiting, assignment)$

only the parameters of the Delta operator has changed

unknownPlane

only the Xi operator is removed

$totalAssignment \mathrel{\widehat=}$
$[plane? : Plane, gate? : Gate] \bullet$
$assignGate$
$[\!]$
$unknownPlane$

The choice operator ($[\!]$) is used in the definition of the operation *totalAssignemt*, allowing the system to choose one of the two alternative operations *assignGate* and *unknownPlane* without user intervention. The variables in square brackets are those for which input values are expected from the system environment. The operator $\bullet$ is used to promote[1], when necessary, operations through the selected objects (in square brackets). This operator has the advantage of providing a way to inherit operations from objects of other classes. The concept of inheritance, discussed below, may be introduced in the definition of a class in different ways.

2.3.2 Inheritance

The concept of inheritance allows for the reuse of features of an inherited class (the superclass) when creating a new class schema (the subclass). As mentioned earlier, Object-Z

[1] **promotion** allows for the reuse of an operation to specify another one

provides different specification constructs to define the inheritance mechanism, e.g. through class inclusion, by using a class as a type or promoting an operation.

(a) ***Class inclusion***
The name of the inherited class is listed in the declaration of the inheriting class. In that case, the type and constants of both classes are merged as well as their schemas. But, state schemas as well as those that share the same name are joined. The visibility list is not inherited.

(b) ***Class schema as a type***
Consider the following declaration allowed in Object-Z where *ClsAirport* is the class defined earlier:

$$orTambo : ClsAirport$$

This definition specifies the variable *orTambo* as an identifier of an object of the class *ClsAirport*. Object identity is modelled in Object-Z by associating with each class name a countable infinite set of values (Smith [77]). Through the variable *orTambo* and the dot (.) notation, the features of the class *ClsAirport* become accessible to the class in which it is declared. E.g. an operation may change the state of the referenced object as follows: $orTambo.waiting' = orTambo.waiting \cup \{plane1\}$, where $plane1$ is of type *Plane*.

(c) ***Operation promotion***
The scope enrichment operator ($\bullet$), the dot and the possibility to use a class as a type in Object-Z provide meaningful ways to specify the reuse of operations. Consider for example the following operation:

$$newAssign \mathrel{\widehat=} [orT? : ClsAirport \mid orT?.waiting \neq \varnothing] \bullet orT?.totalAssignment$$

The operation *newAssign* in a class, is defined by promoting the operation *totalAssignment* of an object of the class *ClsAirport* referenced by orT.

The concept of polymorphism is briefly discussed in the following section.

2.3.3 Polymorphism

In Object-orientation, the concept of polymorphism defines a mechanism which allows a variable to be declared, whose value can be an object from any of a given collection of

classes. In Object-Z, polymorphism is introduced with the unary class operator denoted by th symbol $\downarrow$, e.g. the declaration

$$orTambo : \downarrow ClsAirport$$

specifies an object of the class *ClsAirport* or any other class derived from it by inheritance.

2.3.4 Tool support for Z and Object-Z

An important advantage of using Z, is the availability of tool supports, allowing for the possibility to reason about the properties of the specification (van der Poll [90]). Z tools include amongst others the following: CadiZ (Toyn and Mcdermid [87]) for formal reasoning, and Fuzz Mike Spivey's type checker for Z. The Community Z Tools (CZT)(Malik and Utting [54]) are used for type-checking and animating Z. Unlike Z, the tools associated to Object-Z are still limited and many of them operate specifically under Linux. Examples are: the latex macro OZ.sty (Allen [3]) for editing Z and Object-Z specifications. The Wizard (Johnston [43]) and the Object-Z version of the Community Z Tools (CZT) (Malik and Utting [54]) for type checking. It has been proposed to encode Object-Z into existing theorem provers (e.g. Smith et al. [80]). A methodology to animate Object-Z specifications using a Z animator (McComb and Smith [56]) and for model-checking Object-Z using Abstract State Machine (ASM) (Winter and Duke [98]) have also been suggested.

2.4 Chapter summary

This chapter presented an overview of the three specification notations used in this dissertation. The semi-formal notation UCMs (Use Case Maps) was initiated by Buhr to facilitate the capturing and analysis of requirements from users, to model an early perception of the static and dynamic behaviours of a system and its architectural structuring of components in map-like diagrams. Z and Object-Z are state-based formal specification languages based on set theory and first order predicate logic. The central concept in Z is the schema, to describe the possible states and operations of a system. In Object-Z the central concept is the class schema to encapsulate Z types and schemas and to introduce object-orientation to standard Z.

The next chapter presents the case study used in this dissertation.

Chapter 3

Case study

This chapter describes a case study that will be used to explore the topic of this research. It also describes the two approaches adopted to develop the case study. A UCM model for the case study is developed and some observations are made regarding the use of UCMs. A Z specification of the same case study is presented including, calculating preconditions for operations and the construction of total operations that are summarised in a table, leading to a conclusion.

3.1 Case study description

Imagine a group of rival companies geographically dispersed world-wide who wish to co-operate. Each of them provides amongst others, sales services and allows for credit, return and replacement of goods purchased (e.g. those under guarantee or warranty). Each company has both local and international customers, and uses its own sales systems. Assume that after some market studies and analysis, the representatives of those companies come to a common conclusion that a very high percentage of their revenue is due to their international customers whose transactions are, nevertheless, very limited because of the difficulty to return or replace items. Additionally, they also realise that those customers incur enormous charges when paying with credit cards or bank transfers, compared to a zero charge of a direct payment at a cashier in a local outlet. In this regards, they decide to help each other, to encourage their international customers and hence, increase their benefits. They came up with an innovative idea of having each company acting at the customers' level, as an agency of any other one, relative to the above-mentioned operations (return items, replace items, and pay credit), provided that the independence and privacy of each individual company should not be violated. That is,

- No individual strategic plan and mission statement should be affected;

- None of the standards and policies adopted by each individual company should be influenced;
- A company should not be forced to operate with a language or currency that it is not used to;
- The organisational structure of a company should not be affected.

Let's further assume that the representatives of those companies believe that such innovation should be software-based and therefore, in order to take further decisions, they need a good system specification that would facilitate their understanding, and stimulate a thorough discussion about the feasibility of the idea, and therefore, help them to discover a possible way to render such an idea operational. Finally, assume that the number of member companies is not limited, and it is agreed to produce an Object-Z specification to serve the purpose.

3.2 Specification approach

To design the specification for this case study, a number of approaches are plausible. For example, a Use Case approach where Use Cases are identified and transformed into an Object-Z specification (Moreira and Araújo [63]). But as one of the main objectives of this dissertation is to explore the impact of using the UCM notation in the construction of Object-Z (including Z), two different specification processes are adopted (see fig. 1.3, Chapter 1) to produce two Object-Z specifications. One approach uses UCMs, and the other not. The two resulting Object-Z specifications are then compared.

- With the first approach, a UCM model for the case study is constructed then, transformed into Z and Object-Z.
- With the second approach, a Z specification of the case study is constructed and transformed into Object-Z.

For both approaches, the informal requirements described above are used as input, first to derive a UCM in Section 3.3, and a Z specification in Section 3.4. The transformation of those intermediary specifications is done in upcoming chapters. The main operations retained in each case consist of returning items, replacing items, and credit payment by customers. When any of those operations is initiated at an agency A, to assist a customer of another agency B, A is named **Helper**, and B, the **Beneficiary**.

It is also assumed that an Interface subsystem is available at each company to facilitate the communication between the system under specification, and a local sale system. Figure 3.1 depicts an example of interconnection between agencies in an operational view. Each

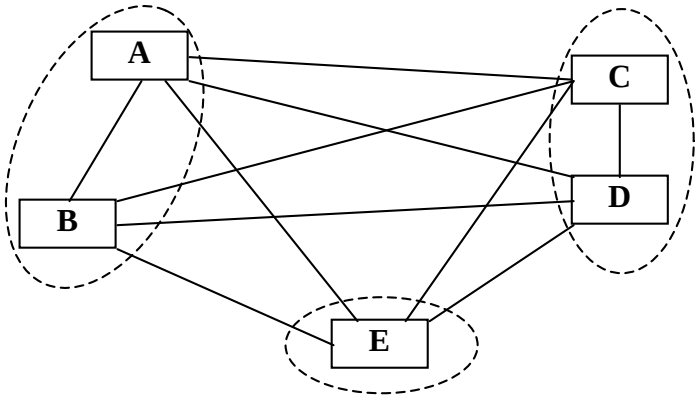


Figure 3.1: Example of agencies interconnections

rectangle represents a company, and the dotted ellipse those companies that are in the same area, e.g. the same country. A solid line joining two agencies indicates a network connection between the two companies. Figure 3.2 gives an overview of the system layering.

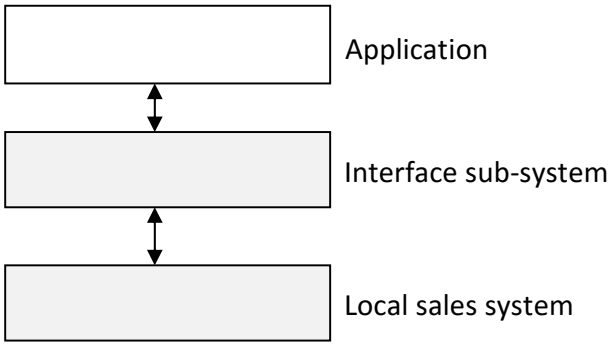


Figure 3.2: Subsystems layering

The Interface and the Local Sales sub-systems are shaded because there are not explicitly part of the system under discussion. A top-down approach is considered as we start with the three given higher-level functionalities, and then refine them continuously, while constructing the specification, until a reasonable level of detail is reached. The refinement process is done according to the ability of the specification technique, to allow such decomposition by providing mechanisms to represent sub-operations in a traceable way. It is assumed that a reasonable level of detail is reached when the execution steps of each scenario are clearly defined, and the overall size of the system is still manageable in the context of this work (with its time and space limitations).

3.3 Deriving a UCM for the case study

This section proposes a Use Case Maps Model for the case study. The construction process is based on guidelines from the literature. Two types of documents are used: a quick tutorial on UCM by Amyot [4], and the UCM book authored by Buhr and Casselman [20], the initiators of UCMs. The scenarios are initially considered at a high-level of abstraction, and progressively refined to include detail activities. Functionalities are first represented then, the architectural components are added to the map.

3.3.1 Initial UCM

Figure 3.3 represents the initial UCM map for the case study. Each high-level functionality is considered a complete scenario. The initial UCM diagram is progressively refined until a "reasonable" level of detail is reached, that is, a map including all the user operations required by the system. The path from the start point **S1** to **E1** models the scenario of returning a

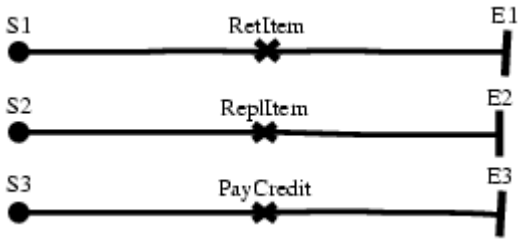


Figure 3.3: Initial UCM

purchased good. This path includes only one responsibility point *RetItem*, which stands for return. Similarly, the path from **S2** to **E2** models the scenario to replace a purchased item, and the path from **S3** to **E3** models the scenario to pay a credit.

Triggering events

Although it is not explicitly stated in the description of requirements, the execution of each of the above scenarios may begin when a customer contacts an agency wanting to pay a credit, or return or replace a purchased good. For the return and replacement of a purchased item, the customer may be required to present an invoice together with the item. A customer's identity document (or any other acceptable personal information document) may also be required for any of the three scenarios.

Path preconditions

At this point, it seems reasonable to suggest that for an item to be replaced or returned, the invoice provided by the customer should be valid. Additionally, the customer must have a valid account in the provider's local sales system.

Resulting events and post conditions

The execution of the scenario to return an item begins when the start point **S1** is triggered and terminates at the end point **E1**. At this point, the returned item has been collected by the provider, and all the accounts that may be affected by the operation are updated. The scenario to replace a purchased item terminates at the end point **E2**, when the execution is successful. At this time, the returned item has been sent to the provider (where traditional routine procedures may be followed to finalise the operation). Finally, when the payment of a credit is successfully performed, the effect at the end point **E3**, is that the Helper agency has received some amount of money from a customer, and all the accounts affected by the operation have been updated.

The simplicity of the map in Figure 3.3 shows that the modelling of the system at this point remains very abstract and hence, keeps some aspects of the system (for example, the sequence of activities for each scenario) hidden. For such reason, a more explicit UCM map, which stands to be an improved version of the previous one, is proposed next.

3.3.2 An improved UCM

Based on the UCM in figure 3.3 and the analysis of the case study description, a more detailed UCM is proposed in Figure 3.4. Large scale responsibilities are refined to allow detail functionalities to be modeled. Consider for instance, a precondition that requires an invoice to be valid to allow the return or replacement of a purchased item. This precondition to be reinforced requires the provider of the item, to first validate the invoice presented to the Helper by the customer. The validation of an invoice is therefore considered an important functionality of the system. Figure 3.4 includes such new functionalities, which improve the diagram in Figure 3.3. The diagram in Figure 3.4 shows both the behaviour (the execution of the sequence of activities) of the system, and the interaction between scenarios. The next two sub-sections provide more detailed explanations on those two aspects, with the purpose of improving the understanding of the UCM in Figure 3.4.

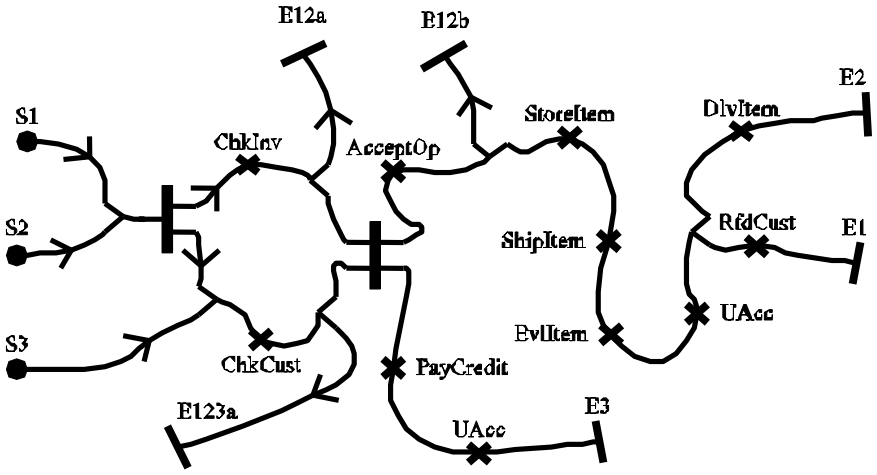


Figure 3.4: Improved UCM

The system behaviour

In general, the execution of a scenario begins when some events occur at some start points and the preconditions at those points are satisfied. As mentioned previously, a customer arriving at an agency with an invoice, a purchased item and a personal identity, may be allowed to return or replace the item. A customer's identifier is a prerequisite for the payment of a credit. At the Helper agency, only a visual inspection of the invoice, item and customer ID may constitute a valid precondition to trigger the start points **S1** and **S2**. Following the payment of a credit, only the customer ID document may be required at the start point S3. The next paragraph considers the scenario to return an item, aiming to clarify the activities of the system along paths. This case covers much of the responsibility points and other path elements encountered on the UCM in Figure 3.4.

When returning an item, after **S1** is successfully triggered, the invoice and the customer's identity are checked to permit further actions. The responsibility points *ChkInv* to validate an invoice, and *ChkCust* to validate a customer's identity, are performed. With the responsibility point *AcceptOp*, the beneficiary agency may allow the operation to continue or not. If the invoice is found to be invalid, the execution reaches the end point **E12a**, and terminates. Similarly, if the validation of the customer fails, the end point **E123a** is reached, causing the scenario to be terminated. When the validation of the invoice and customer succeed, and the operation is not denied by the provider, the item brought by the customer is temporally kept in a store (*storeItem*) at the Helper agency waiting to be shipped back *shipItem*. At the provider's agency, the returned item is collected and evaluated *EvlItem* to determine its present value. At this point, the beneficiary agency follows their traditional procedures to

update the customer's account (*UAcc*), and, for example, refund the customer (*RfdCust*). The execution of the scenario is terminated at the end point $E1$.

The progression of the other two scenarios may be interpreted in the same vein. The only two operations not encountered, are *DlvItem* that may be performed when replacing an item to provide a substitute item to a customer; and *PayCredit* to allow a customer to pay some amount of money at the Helper agency which is then transferred to the benefiary agency. The sub-section below illustrates some interactions between scenarios as depicted in the UCM model of Figure 3.4.

Scenario interactions

Interactions between the three scenarios are observable from the UCM in Figure 3.4. For example, the fact that the scenarios to return a purchased item, and the one to replace an item, both share most of their path segments and responsibility points, shows that these two scenarios are closely related. This observation indicates that the procedure to perform each of the scenarios may be very similar. The analysis of scenario interactions presents the advantage to facilitate the understanding of the system, and guides further design decisions. For example, the above observation readily suggests a strong use of polymorphism.

Contrary to the above observation, the procedure to perform the scenario to pay a credit is clearly very different from the others. The three scenarios share in common only one responsibility point ("ChkCust"), on a single path segment. More interactions may be analysed by considering other path elements, e.g. path connectors: OR-join, OR-fork, AND-fork and AND-join. This analysis, at an early stage of the requirement specification, also presents the advantage to facilitate the detection of conflict points in the system. For example, since the responsibility point "ChkCust" is shared by all the scenarios, more attention may be required at the design of this operation.

The following section presents a more detailed version of the UCM that includes abstract components.

3.3.3 A more detailed UCM

Although the UCM map in Figure 3.4 adds more detail into the previous diagram, it still does not reveal important information that needs to be addressed explicitly. For instance, the validation of an invoice or a customer, also involves sending a request, over a communication network (see fig.3.1), to the benefiary agency and waiting for a response. The role

of the three parties: Helper agency, Beneficiary agency, and the Network are not distinctly illustrated in such operations. To address this issue, in Figure 3.5, architectural elements are introduced into the map and some responsibility points are sub-divided into separate sub-activities. In this last version of the map, the team components named respectively "Helper"

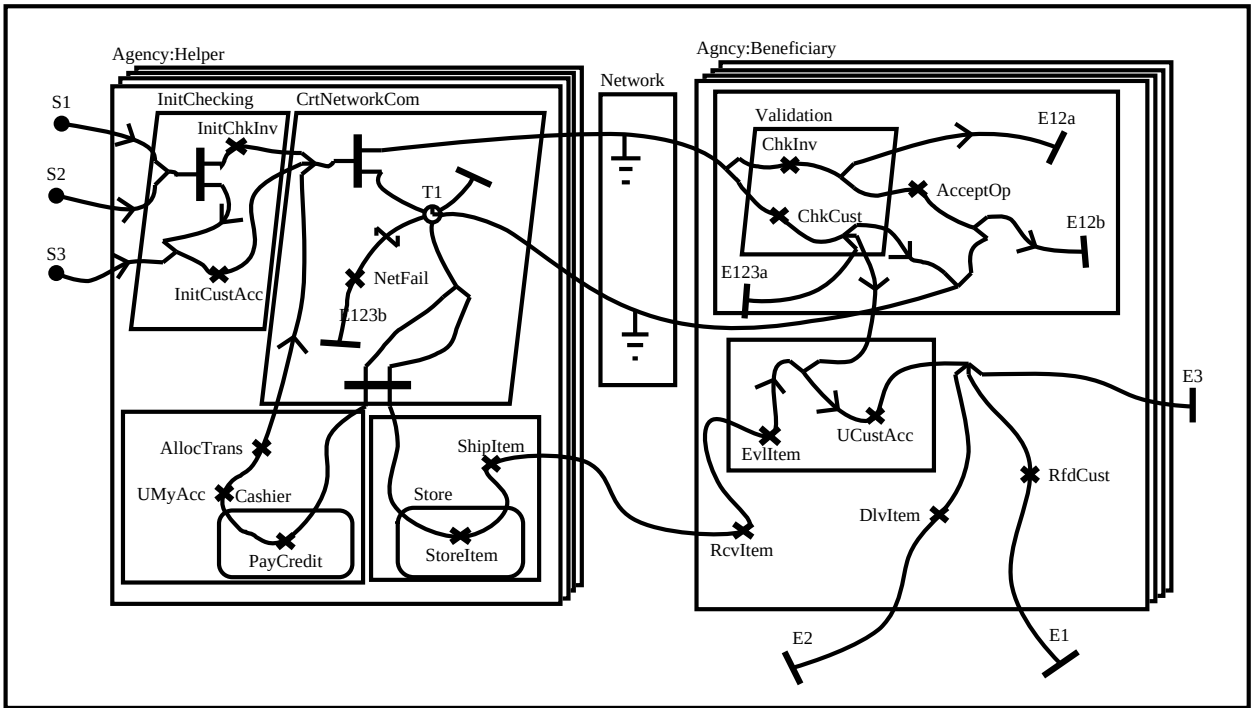


Figure 3.5: Final UCM

and "Beneficiary" logically sub-divide the system into two sub-systems. The first includes the activities performed in an agency when acting as a Helper, and the second, groups those activities that are executed in an agency, when playing the role of a Beneficiary agency. Any agency may play both roles; it acts as a helper when helping a customer, and as a beneficiary when its customer is being helped by another agency. As more than one agency is involved, many identical copies of the components can be superimposed to indicate the multiplicity of the sub-systems.The component named Network, in between the two sub-systems, handles any electronic communication between the two sub-systems.

The next two sub-sections explain each of the sub-systems.

Helping sub-system

This sub-system includes all the activities that are performed in an agency when acting as a helper. Those activities are structurally grouped into two abstract components (processes): "InitChecking" and "CrtNetworkCom", and two team components (see Figure 3.5).

The process named *InitChecking* has the responsibility to handle the triggering of scenarios at the start points, and prepare requests to be forwarded over the network to demand, for example, the validation of an invoice, or a customer, by the Beneficiary sub-system. It therefore performs "InitChkInv" to initialise the validation of an invoice and "InitCustAcc" to initialise the validation of a customer. It also controls scenario interactions, that are modeled with path elements placed within it; those are the OR-join path connector that introduces path sharing, and the AND-join connector, which enables the concurrent execution of the two operations placed before it.

The process *CrtNetworkCom* has the responsibility to control the incoming and outgoing network communications. It transmits the requests prepared by *InitChecking* over the network and waits for responses using the UCM component timer (T1 on Figure 3.5). With the timer, it implements a timeout-recovery mechanism (see Buhr and Casselman [20]) to resolve network failures. Any incoming request is forwarded to the network component and the timer is set-up with an appropriate value, which represents the maximum waiting time for a response to arrive. If the time elapses before any response is received, failure is assumed, and the responsibility point "NetFail" is performed, and the execution of the scenario is terminated at the end point E123b.

Similarly to *InitChecking*, the process is responsible for controlling interactions between UCM path segments that are bound to it. For example, path sharing is introduced by the OR-join connector, and parallel execution of responsibility points introduced by the AND-fork connector. The execution of a scenario to return or to replace an item, may continue after the AND-fork bar, only when the invoice, together with the customer, have been successfully validated. Beyond this component, an appropriate path segment is followed depending on the scenario being executed.

After a customer has paid an amount of money at a cashier in a Helper agency (*PayCredit*), the system updates the account of the beneficiary agency (*UMyAcc*) and sends a request to demand the beneficiary agency to update the customer's account (*AllocTrans*). An object component is used to model a cashier and a store. These objects are each placed in a component that controls their activities.

Beneficiary sub-system

This sub-system aims to assist Helper agencies in validating invoices (*ChkInv*) and customers (*ChkCust*), and to authorise (*AcceptOp*) a return or a replacement of a purchased item. Cus-

tomers' accounts and profiles are locally managed by a local sales system (see Figure 3.2) to which external agencies do not have direct access. It also assumes the responsibility to finalise any operation initiated at a helper sub-system. With a credit payment, it updates the customer's account on request, and ends the scenario at the end point E3. In the case of the other two scenarios, it collects the item (*RecvItem*) shipped from another agency, evaluates the item (*EvlItem*) and Updates the customer's account (*UCustAcc*) accordingly. Then, it performs some local routines to either refund the customer (*RfdCust*), or to deliver another item to the customer (*DlvItem*). Some of those local routines may not be integrated or known by this system, because its purpose is to facilitate the liaison between customers and their providers (agencies).

This sub-system is structurally composed of one UCM team component, that includes two other team components. One process that ensures the validation tasks, authorises remote operations, and controls premature terminations of scenarios such as terminating a scenario when the validation fails. Another process evaluates returned items and updates customers' accounts.

The next section presents in a textual form, scenarios as described in the UCM Model of Figure 3.5.

3.3.4 Scenarios

A number of UCM traversal techniques that can help to extract scenarios from a UCM diagram have been proposed (e.g. Amyot et al. [9], Kealey and Amyot [47]). This section presents a textual description of scenarios as described on the map in Figure 3.5 to explain the clarification brought so far, by UCM, to facilitate the comprehension and the description of the case study. Naturally such textual description may not reveal all the important aspects of the map. It does not, for instance, address interactions between scenarios, path sharing, parallel progression of scenarios along paths, and is largely silent about the architectural structure of a system.

(a) Return item

Description: this scenario describes the sequence of steps that are followed, in an agency, e.g, Helper, to help a customer to return an item purchased from another agency, e.g., Beneficiary.

S1: A customer comes to the agency wanting to return an item. The customer holds an invoice, the item to return, and a personal identifier document.

1. *InitChkInv*. Helper initialises the invoice checking and sends a request, via the network, to the Beneficiary to validate the invoice.
2. *InitCustAcc*. Helper initialises the authentication of the customer, and sends a request to the Beneficiary to check if the customer holds a valid account.
3. $T1$. The Helper sets up a timer, forwards the request through a network, and waits for a response from the Beneficiary.
4. *ChckInv*. The Beneficiary validates the invoice.
5. *ChckCust*. The Beneficiary validates the customer's account.
6. *AcceptOp*. The Beneficiary allows or denies the operation and sends a result back to the helper via the network.
7. *StoreItem*. The Helper receives the item from the customer and keeps it temporarily waiting to be forwarded.
8. *ShipItem*. The Helper ships the item to the Beneficiary.
9. *EvlItem*. The Beneficiary evaluates the present value of the item.
10. *UcustAcc*. The Beneficiary updates the customer's account.
11. *RfdCust*. The Beneficiary, locally, uses its own routine procedure to satisfy the customer outside the system.

Alternatives:

1. *NetFail*. Timeout event occurs before any response from the Beneficiary, the Helper performs *NetFail* to manage the failure.
2. $E12b$. The Beneficiary denies the operation; the path segment to E12b is followed to terminate the scenario.
3. $E12a$. The invoice is not valid; the path to E12a is followed to terminate the scenario.
4. $E123a$. The customer does not have any account with the beneficiary; the path segment to E123a is followed to terminate the scenario.

(b) Replace item

Description: this scenario describes the sequence of steps that are followed, in an agency e.g. Helper, to help a customer to replace an item purchased from another agency e.g. Beneficiary.

S2: A customer comes to the agency wanting to replace a purchased item. The customer holds an invoice, the item to be replaced and a personal identifier document.

According to the model, the sequence of steps for this is similar to those of the previous one. Only the routine tasks, performed locally to satisfy the customer that is outside the scope of the system, are different. Alternative scenarios are also identical. We then, refer the reader to the above-description.

(c) Pay credit

Description: this scenario describes the sequence of steps followed, to help a customer to pay for a credit at a company e.g. Helper, to the benefit of another company e.g. Beneficiary.

S3: A customer comes to the Helper wanting to pay for a credit. The customer holds a personal identifier document (or a reference number of the account at the Beneficiary company).

Sequence of steps:

1. *InitCustAcc*. Helper initialises the invoice checking and sends a request, via the network, to the Beneficiary to validate the invoice.
2. $T1$ The Helper sets up a timer, forwards the request through a network, and waits for a response from the Beneficiary.
3. *ChckCust*. The Beneficiary validates the customer's account.
4. *PayCredit*. The Helper receives cash from the customer at the cashier.
5. *UmyAcc*. Helper Updates the common account associated to the cashier.
6. *AllocTrans*. Helper allocates the transaction into the Beneficiary account kept locally in the system, and sends a summary of the transaction to the beneficiary through a network connection.
7. *UcustAcc*. The Beneficiary updates the customer's account.

Alternatives:

1. *NetFail*. A timeout event occurs before any response from the Beneficiary is received; the Helper performs *NetFail* to manage the failure.
2. $E123a$. The customer is not recognised (an invalid customer) the path segment to E123a is followed to terminate the scenario.

Referring to the literature and the previous UCM specification experience, the following section addresses some important issues on UCMs. They are presented as observations.

3.3.5 Some observations on UCMs

Arguably, UCMs do not, sufficiently, formally define certain aspects of a specification, for example, UCMs do not provide for a means to calculate preconditions for paths, path segments or responsibility points. However, the notation presents a number of advantages. The core of these advantages stems from the fact that it conveys different type of information, on complex systems, in a map-like diagram, using simple graphical elements; and is mainly human oriented. The following are some useful observations that may be inferred from the experience of the above case study guided by insights from the literature.

1- A UCM facilitates the capturing and description of scenarios (Amyot [4]). It is flexible in terms of allowing the modification of the existing model to include changes in requirements. This may be observed in the incremental construction process that is used in the specification of the case study; moving from the initial UCM of Figure 3.3 to that of Figure 3.5. The method allows users to change, delete path elements, change their position in the map, as well as to add new elements. Similarly, abstract components that are also graphical elements can be manipulated.

2- The UCM techniques may offer the possibility, as illustrated in the previous observation, to explore different ways of grouping the functionalities of a system with abstract components to yield an appropriate architectural structuring of the system and hence, the final UCM may constitute an important input for the design and analysis of system components .

3- The technique provides a "global view" of a system, in a map like-diagram, including scenarios, scenario interactions, and structural organisation (see Figure 3.5). Such a view may facilitate the analysis of the system as a whole, as well as the analysis of individual scenarios at an earlier stage, as it aims to represent the intended picture of an overall system (e.g. Buhr and Casselman [20], chap.2). It may also make it easier to simulate the execution of a system at an early stage, resulting for instance, in speeding-up the process of building prototypes, specification animation, and consequently the detection of some potential problems in requirements such as inconsistencies, missing requirements, undesirable effects of scenario interactions, bottlenecks in scenario coupling, etc.

   For example, referring to the diagram in Figure 3.5, knowing that the operation *UCustAcc* is solicited by all scenarios, and that, at some stage, multiple instances of such scenarios operate concurrently during the operational phase, could positively influence some design decisions.

4- The construction and manipulation of a UCM model does not require extra effort from

a specifier who has a basic understanding of the concepts used in requirements engineering (e.g. Requirements, Use Cases, Scenarios, etc.). The two main construction tools: UCMNav and JUCMNav are also downloadable from the Internet free of charge (URL:***http://www.UseCaseMaps.org/***). To explore their potential, both of them were used in this dissertation. JUCMNav is more recent and includes more functionalities than UCMNav.

5- Although the notation does not have a formal semantic definition, its formal syntax, and the fact that the basic notation is based on simple graphical elements, make it transformable into other models. A number of transformations have been proposed (e.g. Bordeleau and Buhr [12], Miga et al. [60], Zeng [102]). This transformability quality, coupled with the above observation on its ease of use may contribute to provide the model with the ability to be used as a "bridging tool" in software specification and design (Dongmo and van der Poll [23]). The notation was initially intended to bridge the gap between user requirements and design (Buhr [17], Buhr and Casselman [20]).

6- The model may also be a potential candidate for feasibility studies and estimations in software project management. Not only can it be flexibly constructed from a "fuzzy" set of requirements, but it also clarifies the understanding of scenarios (as illustrated in Section 3.3.4) in large and complex systems (Buhr [18], Buhr et al. [21]).

The next section constructs a Z specification for the case study.

3.4 Z specification

In practical projects, multiple source of information may be available. Those include, for instance, users, clients, domain experts, etc. The present specification relies on our understanding of the case study description and is guided by the established strategy for constructing a Z document as given in a number of texts on Z (e.g. Lightfoot [52], Potter et al. [70]). When necessary, use will also be made of some principles suggested by van der Poll and Kotzé [93] to reinforce the original Z strategy. In line with principle no# 4, [93], which recommends extending each set to include undefined outputs, an assumption is made that types in this specification readily include the undefined value that is denoted by the symbol $\perp$. Promoting operations [1] is avoided, wherever possible, at this stage of the specification, since this will be included in the Object-Z specification version (Z-OZ) through inheritance. More detail on operation promotion, and framing in Z are found in: Woodcock and Davis [101, Chapter 13] and Stepney et al. [84].

[1] That is to extend an operation defined on a smaller state schema to be used in larger one

Next a Z specification of the case study is provided. It starts with the basic types.

3.4.1 Given sets and global variables

By convention, each element of the list is in singular, with the first letter capitalised. The list of basic types is:

$$[Item, Customer, AgencyId, Invoice, Currency, Transaction, Accountno, Money, Language, Report, Address, Date]$$

The list was progressively constructed during the specification process, by adding new types when needed. The following is a brief description of the listed types:

Item is the set of all possible items that exist.

Customer is the set of all possible customers.

AgencyId is the set of all possible identifiers of agencies.

Invoice is the set of all invoices.

Currency is the set of all possible currencies.

Transaction is the set of all possible transactions.

Accountno is the set of all possible account numbers.

Money is the set of all possible amounts of money.

Language is the set of all possible human languages.

Report is the set of all possible messages that may be exchanged with the system.

Address is the set of all possible addresses.

Date represents all possible dates.

As different currencies may be used in the system, the function *exchange* is defined to exchange money from one currency to another.

$$exchange : Money \times Currency \times Currency \pfun Money$$

Similarly, the system may need to translate a message from one language to another; the function *translate* is defined to serve this purpose.

$$translate : Message \times Language \times Language \pfun Message$$

A transaction is defined as an object from which the value can be extracted using the function amount.

$$amount : Transaction \pfun Money$$

To optimise the communication with users (van der Poll and Kotzé [93, principle no.2]), a number of possible Report Messages are defined. Each message reports on the success or failure of an operation.

$$Success,$$
$$ItemAlreadyReturned,$$
$$UnknownIdentifier$$
$$AgencyNotFound,$$
$$InvalidInvoice,$$
$$ItemAlreadySent,$$
$$ItemNotReceived,$$
$$IncorrectAddress,$$
$$TransactionNotFound,$$
$$UnknownCustomer,$$
$$AgencyAccountNotFound,$$
$$PaymentNotFound,$$
$$TransactionNotAllocated$$

$$\langle Success, ItemAlreadyReturned, UnknownIdentifier, AgencyNotFound,$$
$$InvalidInvoice, ItemAlreadySent, ItemNotReceived, IncorrectAddress,$$
$$TransactionNotFound, UnknownCustomer, AgencyAccountNotFound,$$
$$PaymentNotFound, TransactionNotAllocated\rangle \in iseqReport$$

The next section presents the abstract state schemas of the system.

3.4.2 Abstract state space

Due to the fact that some operations in the case study involve cash payments, an abstract state is defined to describe the object *accounts*, to facilitate such operations.

Account

$$accountno : Accountno$$
$$balance : Money$$

$$account \neq \bot$$

An object of type *Account* is uniquely referenced with an *accountno*. It contains a variable *balance* to specify the balance of the account, which value is meaningless when the *accountno* is undefined.

The following abstract state specifies the state of the communication interface between the system under-consideration and the Local Sales System in an agency (see Figure 3.2).

$$
\begin{array}{|l}
\textit{ISales} \\
\hline
\textit{custaccounts} : \textit{Account} \rightarrowtail\!\!\!\!\!\!\!\!\rightarrow \textit{Customer} \\
\textit{statements} : \textit{Account} \times \textit{Transaction} \nrightarrow \textit{Date} \\
\textit{invoices} : \mathbb{P}\, \textit{Invoice} \\
\hline
\mathrm{dom}(\mathrm{dom}\, \textit{statements}) \subseteq \mathrm{dom}\, \textit{custaccounts} \\
\hline
\end{array}
$$

The variable *custaccounts* represents the list of all customer accounts in the local system, that are made accessible to other agencies. The variable keeps the set of all the transactions made on a customer's account. The set of all the invoices are maintained in the variable, *invoices*. Only information made accessible from outside the local sales system is contained in the state space, *ISales*. As indicated in the predicate, transactions are recorded only for those customers who have an account in the local system. The state schema that follows describes a database containing essential components that may reveal the status of the system, relative to the three major services - return, replace items, and pay credits - of the system (see Section 3.1 and Figure 3.1).

The schema *Database* is relevant at the level of an agency. It includes a list of references (*agencies*) to those agencies that are part of the system. A unique account is also created for each individual agency to record the balance of all its transaction (*agencyaccounts*). The component *itemsin*, records all the items received from customers, and *itemsout* maintains a set of items received from customers and sent out to the beneficiary agency. A *cashin* specifies a cashier where all payments are made at an agency. For each payment operation, a transaction is created and mapped to the corresponding account (*statements*). The component *collected* captures all the returned items forwarded by other agencies (Helpers).

$$
\begin{array}{l}
\textit{Database} \\
\hline
agencies : \mathbb{P}\, Agencyid \\
agencyaccounts : Account \rightarrowtail\!\!\!\!\!\!+ \; Agencyid \\
itemsin : Invoice \times Item \times Date \mathrel{\ooalign{$\rightarrow$\cr\hidewidth$+$\hidewidth}} Agencyid \\
itemsout : Item \times Date \mathrel{\ooalign{$\rightarrow$\cr\hidewidth$+$\hidewidth}} Address \\
collected : Item \times Agencyid \mathrel{\ooalign{$\rightarrow$\cr\hidewidth$+$\hidewidth}} Date \\
cashin : Customer \times Money \times Date \mathrel{\ooalign{$\rightarrow$\cr\hidewidth$+$\hidewidth}} Agencyid \\
statements : Account \times Transaction \mathrel{\ooalign{$\rightarrow$\cr\hidewidth$+$\hidewidth}} Date \\
\hline
\mathrm{ran}(agncyaccounts) \subseteq agencies \land \mathrm{ran}(itemsin) \subseteq agencies \\
\mathrm{dom}(\mathrm{dom}\, itemsout) \subseteq \mathrm{ran}(\mathrm{dom}(\mathrm{dom}\, itemsin)) \\
\mathrm{ran}\, cashin \subseteq \mathrm{ran}\, agencyaccounts \\
\mathrm{dom}(\mathrm{dom}\, statements) \subseteq \mathrm{dom}(agencyaccounts) \\
\mathrm{ran}(\mathrm{dom}\, collected) \subseteq agencies \\
(\forall\, Account \mid \theta Account \in \mathrm{dom}(statements)) \bullet \\
\quad (\exists\, id : Agencyid \bullet id \in \mathrm{ran}(cashin) \land \theta Account \mapsto id \in agencyaccounts)
\end{array}
$$

As indicated in the predicate, a service is rendered for a Beneficiary agency only when an account and a valid identifier are created for the agency. The system requires that any item shipped to a provider must have been received from a customer. The last predicate indicates that transactions recorded in *statements*, capture exclusively payments made at a cashier for Beneficiary agencies.

The next schema describes the state of an agency, which includes the above schema.

$$
\begin{array}{l}
\textit{Agency} \\
\hline
Database \\
identifier : Agencyid \\
dcurrency : Currency \\
address : Address \\
language : Language \\
ssales : ISales \\
\hline
identifier \notin agencies \\
\mathrm{dom}(ssales.custaccounts) \cap \mathrm{dom}(agencyaccounts) = \varnothing
\end{array}
$$

Additionally to the data provided by the schema *Database*, some information is added to personalise each agency. The variable *dcurrency* contains the currency that can be used from outside the agency. Similarly, *address* and *language* represent respectively the address and language that can be used to communicate with the agency. The unique reference of the agency is recorded in *identifier*. Each agency provides to others a unique interface *ssales* to communicate with its local sales system. The sharing of accounts is not allowed.

The system itself is modeled in the next schema as a set of agencies (see Figure 3.1).

System

$known : \mathbb{P}\, Agency$

The next section proposes realisable initial states for the above state schemas.

3.4.3 Initialising the state space

This section presents for each abstract state schema previously defined, an initial state that is assumed. Although a formal proof needs to be provided according to the established strategy for deriving Z documents, such proofs are omitted. However, the following initialisation theorem may be followed, when necessary, to establish that each assumed initial state is realisable.

$\vdash State' \bullet InitState.$

Initially the balance of an account is assumed to be zero, and the database is empty.

InitAccount

$Account'$

$balance' = 0 \land accountno' \neq \bot$

InitDatabase

$Database'$

$Agencies' = \varnothing \land agencyaccounts' = \varnothing \land itemsin' = \varnothing$
$itemsout' = \varnothing \land cashin' = \varnothing \land statements' = \varnothing \land collected' = \varnothing$

Initially, the database in an agency is at its initial state, and the interface of communication is assumed empty.

InitAgency

$Agency'$
$Init\_Database$

$ssales'.custaccounts = \varnothing \land ssales'.statements = \varnothing \land ssales'.invoices' = \varnothing$

InitSystem

$System'$

$known' = \varnothing$

Next, are presented the partial operations of the system.

3.4.4 Partial operations

This section describes the partial operations of the system. Those are the operations that model the activities of the system under normal circumstances without considering errors that may occur. The focus at this stage is to identify and describe the essential operations of the system. The Z established strategy for deriving Z documents, clearly indicates the structure of a Z specification (Potter et al. [70]) and how schemas can be constructed (van der Poll and Kotzé [93]). The difficult part is therefore more on how to identify operations and objects of a system to be specified, than to worry about how to build their schemas. In general, some case studies on Z (Bowen [13]), introduce the operation, explain what it does, define the schema and complete the description with a prose text. In this work, the operations that follow are based on a dissection of the case study description in Section 3.1 and our understanding of the major services to be rendered by the system.

Receiving an item from a customer

A customer returns an item to an agency, whereupon the item is temporarily kept in a store waiting to be forwarded to the original provider of the returned item.

receiveItemOk

$$
\begin{array}{l}
\Delta Agency \\
item? : Item;\ inv? : Invoice;\ id? : Agencyid;\ date? : Date \\
addr! : Address;\ lang! : Language;\ resp! : Report \\
\hline
item? \notin \mathrm{ran}(\mathrm{dom}(\mathrm{dom}\, itemsin)) \land id? \in agencies \\
(\exists\, Agency \bullet \\
\quad \theta Agency.identifier = id? \land inv? \in \theta Agency.ssales.invoices \land \\
\quad addr! = \theta Agency.address \land lang! = \theta Agency.language) \\
itemsin' = itemsin \cup (inv?, item?, date?) \mapsto id? \land resp! = Success
\end{array}
$$

For this operation to be performed, values must be provided for the input variables decorated with "?". That is, for example, information about the returned item ($item?$), the identifier of the company that provided the item ($id?$), etc. The system uses the input $id?$ to determine the beneficiary, to validate the invoice ($inv?$) and obtain the necessary information to facilitate the communication: agency's address ($addr!$) and local language ($lang!$). This information is specified for the user, and the item is temporarily kept at the "Helper" agency. The operation is allowed only when the customer's provider is part of the system ($id? \in agencies$).

Items temporarily kept in stores are to be forwarded to their final destination. The following operation serves this purpose.

Sending an item back to the provider

The operation is executed to forward an item previously received to its "beneficiary".

sendItemOk

$$
\begin{array}{l}
\Delta System \\
item? : Item;\ dateout?, datecollected? : Date \\
addr? : Address;\ id? : Agencyid;\ resp! : Report \\
\hline
item? \in \mathrm{ran}(\mathrm{dom}(\mathrm{dom}\ itemsin)) \\
(\exists\, inv : Invoice;\ datein : Date) \bullet (inv, item?, datein) \mapsto id? \in itemsin \\
itemsout' = itemsout \cup (item?, dateout?) \mapsto addr? \\
\exists\, Agency \in known \bullet \theta Agency.identifier = id? \land \theta Agency.addr = addr? \\
\theta Agency.collected' = \theta Agency.collected \cup (item?, id?) \mapsto datecollected? \\
resp! = Success
\end{array}
$$

As indicated in the previous operation, input variables are decorated with a question mark symbol (?) and output variables are decorated with the exclamation mark symbol (!). Only items previously received from customers can be sent. A record is kept for items that are shipped to their provider with the date of the operation (*dateout*?). The operation succeeds when the item is collected and acknowledged by the beneficiary agency.

After collecting the item, the agency may then use its own local routine procedure to complete the transaction. However, our opinion is that the system under consideration may need to update the customer's account with an amount of money equivalent to the value of the returned item, as described next.

Refunding a customer

This operation updates a customer's account with an amount of money equivalent to the present value of a good returned by the customer. Due to the fact that different companies may have different management policies, the idea of updating a customer's account does not necessarily mean the customer will be refunded in cash. It may serve as guideline for further decisions. For example, it may help to choose a replacement item.

This operation relies on the communication interface to be visible to other agencies, and through them to the customer, which makes it possible for the customer to access the information from any agency.

refundCustOk

$\Delta ISales$
$amount? : Money;\ cust? : Customer;\ date? : Date$
$resp! : Report$

$\exists\, trans : Transaction \bullet$
$\quad trans \notin \text{ran}(\text{dom}\, statements) \land amount(trans) = amount?$
$\quad \exists\, Account \bullet$
$\quad\quad \theta Account \mapsto cust? \in custaccounts$
$\quad\quad statements' = statements \cup (\theta Account, trans) \mapsto date?$
$resp! = Success$

The value of the input *amount*? is assumed to be the present value of a returned item if the operation is performed when a customer is returning or replacing a purchased item. The system creates a new transaction of the value of *amount*? and applies it to the customer's account. The system communicates with the Local Sales System of the beneficiary agency to identify the customer's account affected by the transaction.

Receiving cash from a customer

This operation registers a credit paid by a customer to a cashier in a "Helper" agency.

receivCashOk

$\Delta Agency$
$cust? : Customer;\ amount? : Money;\ id? : Agencyid;\ date? : Date$
$resp! : Report$

$\exists\, Agency \mid \theta Agency.identifier = id? \bullet$
$\quad \exists\, Account \bullet \theta Account \mapsto cust? \in \theta Agency.ssales.accounts$
$cashin' = cashin \cup \{(cust?, amount?, date?) \mapsto id?\}$
$resp! = Success$

The input variable *cust*? captures information about the customer, *amount*? the amount of money paid by the customer, *id*? the identifier of the beneficiary agency, and *date*? the date of the operation. The system reports the success of the operation with the output variable *resp*!. The system uses the communication interface with the Local Sales System (see Figure 3.2) to ensure that the customer has a valid account with the provider company before proceeding with the payment at the cashier where details of the transaction ar recorded.

After a successful payment, it is suggested the transaction be transferred into the account of the target agency. The operation *allocateTransOk* is defined to serve this purpose.

Allocating a payment to an agency's account

The input *date*? represents the date of this operation and not the date of the earlier transaction made by the customer at the cashier.

allocateTransOk

$$
\begin{array}{l}
\Delta Agency \\
cust? : Customer;\ id? : Agencyid;\ date? : Date \\
resp! : Response \\
\hline
(\exists\, Account;\ trans : Transaction;\ date : Date) \bullet \\
\quad (cust?, amount(trans), date) \mapsto id? \in cashin \land \\
\quad \theta Account \mapsto id? \in agencyaccounts \\
statements' = statements \cup (\theta Account, trans) \mapsto date? \\
resp! = Success
\end{array}
$$

The system uses the inputs containing information about the customer *cust*? and the beneficiary agency identifier *id*? to determine the payment made by the customer at a cashier. A transaction is then created to permanently record the payment in the provider's account at the Helper agency. When this operation is successfully performed, a notification is sent to the Beneficiary company. In this regard, the following operation (*notifyCustTransOK*) is defined:

Notification of payment

The system uses this operation to update, via the communication interface, a customer's account at the "beneficiary" agency after the customer has made a payment.

notifyCustTransOk

$$
\begin{array}{l}
\Delta System \\
cust? : Customer;\ trans? : Transaction;\ datenotice? : Date;\ id? : Agencyid \\
resp! : Report \\
\hline
(\exists\, Agency \mid \theta Agency \in known;\ \exists\, acc, custacc : Account;\ date1, date2 : Date) \bullet \\
\quad (cust?, amount(trans?), date1) \mapsto id \in cashin \land \\
\quad (acc, trans?) \mapsto date2 \in statements \\
\quad \theta Agency.identifier = id? \land acc \mapsto id? \in agencyaccounts \\
\quad custacc \mapsto cust? \in \theta Agency.ssales.custaccounts \\
\quad \theta Agency.ssales.statements' = \\
\quad (\theta Agency.ssales.statements \cup (custacc, trans?) \mapsto datenotice) \\
resp! = Success
\end{array}
$$

The input variable *trans*? captures the transaction allocated to the beneficiary agency's account at the "Helper" side, and *datenotice*? models the notification date. The system

determines all the information recorded on the transaction and uses the communication interface with the Local Sales System at the Beneficiary agency to determine the customer's account and submits the notification. A notification includes information about the customer's account, all the information on the transaction, and the date of the notification (*datenotice*).

Note, that at this stage, the nature of some operations are kept rather abstract. For example, allocating a transaction to an account does not specify explicitly whether the account is debited or credited.

As prescribed by the Z established strategy, preconditions need to be calculated for important operations of the system. This may help to determine circumstances under which an operation is likely to be problematic. Calculating preconditions for the above partial operations is therefore the object of the following section.

Preconditions and total operations

A precondition is an operation, denoted by *pre*, that applies to operation schemas. As advocated by Woodcock [100], a "specifier" has a vital responsibility to ensure the correct precondition for each robust operation that changes the state of a system. Calculating a precondition of an operation helps to describe precisely the conditions under which the operation is applicable, and therefore helps to avoid applying operations outside their domain. Such calculation involves, in general, two major steps (see Potter et al. [70], Woodcock [100], Woodcock and Davis [101]):

- First, to define the precondition schema of the operation by removing the after-state variables and outputs from the declaration part of the operation schema and existentially quantifying them in the predicate.
- Secondly to simplify the schema by applying predefined inference rules and techniques, such as the one-point-rule, which is defined later in this section.

To be realistic, the simplification process may be further deconstructed in multiple steps depending on the complexity of the particular case under consideration; van der Poll and Kotzé [92] illustrate this with an example.
Next, the calculation of the precondition for the partial operation *receivItem*, is considered in detail.

Operation *receivItem*:

Define *Pre receivItem* $\widehat{=}$ *preReceivItem* represented with the schema below:

preReceivItem

$Agency$

$item? : Item;\ inv? : Invoice;\ id? : Agencyid;\ date? : Date$

$(\exists\, Agency';\ ag : Agency;\ addr! : Address;\ lang! : Language;\ resp! : Report) \bullet$
$item? \notin \mathrm{ran}(\mathrm{dom}(\mathrm{dom}\, itemsin)) \land id? \in agencies \land ag.identifier = id?$
$inv? \in ag.ssales.invoices \land addr! = ag.addr \land lang! = ag.lang$
$itemsin' = itemsin \cup (inv?, item?, date?) \mapsto id?resp! = Success$

The schema *preReceivItem* would be extremely hard to use if not simplified. Before attempting to simply this precondition schema, observe that the state invariant still holds for both the before state *Agency*, and the after state $Agency'$. It may also be observed that the quantified variables in the after state that are not changed by the operation, may be omitted in the precondition schema. A formal justification of the last observation is presented shortly, but first, the statement of the one-point-rule used in the proof (e.g. Potter et al. [70]):

If we have an existentially quantified statement, part of which gives us an exact value for the quantified variable then the quantification can be removed, replacing the variable by its known value wherever it appears.

This rule can be translated as follows:

If x is not free in t, then $(\exists\, x : S \bullet P(x) \land x = t) = P(t)$.

Now the proof:
If a predicate $p(x)$ is a tautology, S a state schema, and S' the after state of S returned by an operation, then,

1. $\exists\, x' : T \in S' \bullet x' = x$ [*Hypothesis*]
2. $\exists\, x' : T \in S' \bullet p(x') \land x' = x$ [$p(x')$ *is* true]
3. $x : T \land p(x)$ [*One Point Rule*]
4. $p(x)$ [*fact as* $x \in S$]

Expanding the reference to Agency' and considering only those variables that are changed by the operation in the predicate part, leads us to:

$(\exists\, itemsin' : Invoice \times Item \times Date \pfun Agencyid;\ ag : Agency;\ addr! : Address;\ lang! : Language;\ resp! : Report) \bullet$
(

1. $item? \notin \text{ran}(\text{dom}(\text{dom}\, itemsin)) \land$
2. $id? \in agencies \land$
3. $ag.identifier = id? \land$
4. $inv? \in ag.ssales.invoices \land$
5. $addr! = ag.addr \land$
6. $lang! = ag.lang \land$
7. $itemsin' = itemsin \cup (inv?, item?, date?) \mapsto id? \land$
8. $resp! = Success$

).

The quantified variable $itemsin'$, and the output variable $resp!$, are given exact values and hence, applying the one-point-rule yields:

$(\exists\, ag : Agency;\ addr! : Address;\ lang! : Language) \bullet$
(
1. $item? \notin \text{ran}(\text{dom}(\text{dom}\, itemsin)) \land$
2. $id? \in agencies \land$
3. $ag.identifier = id? \land$
4. $inv? \in ag.ssales.invoices \land$
5. $addr! = ag.addr \land$
6. $lang! = ag.lang \land$
7. $itemsin \cup (inv?, item?, date?) \mapsto id? \in Invoice \times Item \times Date \pfun Agencyid$

).

Each agency is uniquely referenced with an identifier, hence an agency is completely defined when its identifier is known, and therefore, the existence of any other component of a defined agency is implied.

$(\exists\, ag : Agency \mid (id? \in agencies) \land (ag.identifier = id)) \bullet$
$(\exists\, addr! : Address;\ lang! : Language \mid addr! = ag.addr \land lang! = ag.lang).$

The conditions numbers 5 and 6 above, may now be removed, and the output variables may also be removed from the quantification, as their exact values are given, rendering the One-Point-Rule applicable.

1. $item? \notin \text{ran}(\text{dom}(\text{dom}\, itemsin)) \land$
2. $id? \in agencies \land$
7. $itemsin \cup (inv?, item?, date?) \mapsto id? \in Invoice \times Item \times Date \pfun Agencyid \land$

$\exists\, ag : Agency \bullet$
(
3. $ag.identifier = id? \land$
4. $inv? \in ag.ssales.invoices$
).

For clarity, unbound conditions have been placed outside the quantified expression. Next, we prove condition number 7

1.	$item? \notin \text{ran}(\text{dom}(\text{dom}\, itemsin))$	[assumption]
2.	$\forall\, i : Item;\ d : Date;\ id : Agencyid \bullet (i, item?, d) \mapsto id \notin itemsin$	[deduction from 1]
3.	$(inv?, item?, date?) \mapsto id? \notin itemsin$	[deduction from 2]
4.	$itemsin \in Invoice \times Item \times Date \nrightarrow Agencyid$	[by definition]
5.	$(inv?, item?, date?) \mapsto id? \in Invoice \times Item \times Date \nrightarrow Agencyid$ $\Rightarrow (condition7.)$	[from 4]
6.	$(inv?, item?, date?, id?) \in Invoice \times Item \times Date \times Agencyid$	[inputs declaration]
7.	$(id? \in agencies \land$ $\exists\, ag : Agency \bullet ag.identifier = id? \land inv? \in ag.ssales.invoices)$ $\Rightarrow (inv?, item?, date?) \mapsto id? \in Invoice \times Item \times Date \nrightarrow Agencyid$	[from 6]

Since the function that maps the triple elements ($inv?$, $item?$, $date?$) to $id?$ is not total, having an instance for the triple elements does not necessary imply that there will always be an identifier $id?$ associated to that instance. The mapping holds under the assumption that the target agency is part of the system and the input invoice is valid.

The simplified version of *preReceivItem* is:

preReceivItem

$Agency$
$item? : Item;\ inv? : Invoice;\ id? : Agencyid;\ date? : Date$

$item? \notin \text{ran}(\text{dom}(\text{dom}\, itemsin)) \land id? \in agencies$
$\exists\, ag : Agency \bullet ag.identifier = id? \land inv? \in ag.ssales.invoices$

Negating this precondition, leads to the error conditions listed next:

Err1: $item? \in \text{ran}(\text{dom}(\text{dom}\, itemsin))$:
A user trying to register an input item that was already received.

ItemAlreadyReturned

$\Xi Agency$
$item? : Item;\ resp! : Report$

$item? \in \text{ran}(\text{dom}(\text{dom}\ itemsin)) \land resp! = ItemAlreadyReturned$

Err2: $id? \notin agencies$:
The identifier of the input agency is not recognised by the system.

UnknownIdentifier

$\Xi Agency$
$id? : Agencyid;\ addr! : Address$

$id? \notin agencies \land resp! = UnknownIdentifier$

Err3: $\forall\, ag : Agency \bullet ag.identifier \neq id?$:
The system cannot determine the agency with the input identifier. That is, the system cannot get connected to the agency with the input identifier.

AgencyNotFound

$\Xi Agency$
$id? : Agencyid;\ resp! : Report$

$id? \in agencies \land \forall\, ag : Agency \bullet ag.identifier \neq id?$
$resp! = AgencyNotFound$

Err4: $inv? \notin ag.ssales.invoices$:
The invoice presented by the customer is not recognised by the provider company.

InvalidInvoice

$\Xi Agency$
$inv? : Invoice;\ id? : Agencyid$
$resp! : Report$

$id? \in agencies \land$
$\exists\, ag : Agency \bullet ag.identifier = id? \land inv? \notin ag.ssales.invoices$
$resp! = InvalidInvoice$

The Next Z schemas calculus expression defines the total operation for *receivItemOk*.

$$ReceivItem \mathrel{\widehat{=}} receivItemOk \lor ItemAlreadyReturn \lor UnknownIdentifier \lor AgencyNotFound \lor InvalidInvoice$$

Following the same process, the preconditions for other operations may be calculated. For those operations, only the simplified versions of their preconditions with error conditions,

are here presented.

Operation: Forwarding an item from one agency, to another agency.
Define pre $sendItem = preSendItem$.
The schema of the simplified version of this precondition is shown below.

preSendItem

$$Agency$$
$$item? : Item;\ dateout?, datecollected? : Date;\ addr? : Address;\ id? : Agencyid$$

$$item? \notin \text{dom}(\text{dom}\ itemsout)$$
$$item? \in ran(\text{dom}(\text{dom}\ itemsin)) \land id? \in \text{ran}\ itemsin$$
$$id? \in agencies \land \exists\, ag : Agency \bullet ag.identifier = id? \land ag.addr = addr?$$

Negating the schema *preSendItem*, yields the following error conditions:

The first error occurs when a user attempts to send the same item more than once.

ItemAlreadySent

$$\Xi Agency$$
$$item? : Item;\ resp! : Report$$

$$item? \in \text{ran}(\text{dom}(\text{dom}\ itemsin)) \land item? \in \text{dom}(\text{dom}\ itemsout)$$
$$resp! = ItemAlreadySent$$

The condition for the error is $item? \in \text{dom}(\text{dom}\ itemsout)$. Another error condition occurs when $item? \notin \text{ran}(\text{dom}(\text{dom}\ itemsin))$: the item was never received.

ItemNotReceiv

$$\Xi Agency$$
$$item? : Item;\ resp! : Report$$

$$item? \notin \text{ran}(\text{dom}(\text{dom}\ itemsin)) \land resp! = ItemNotReceived$$

The error case where the agency cannot be determined was described previously with the schema *AgencyNotFound* (*Err*2). The next operation schema handles the case of an incorrect address that occurs when the address provided by the user, does not match the real address of the target agency.

IncorrectAddress

$$\Xi Agency$$
$$addr? : Address;\ id? : Agencyid$$
$$resp! : Report$$

$$id? \in agencies$$
$$\exists\, Agency \bullet \theta Agency.identifier = id? \land \theta Agency.addr \neq addr?$$
$$resp! = IncorrectAddress$$

The schema calculus expression that specifies the total operation for forwarding items is:

$$SendItem \mathrel{\widehat=} sendItemOk \lor ItemAlreadySent \lor ItemNotReceiv \lor AgencyNotFound \lor IncorrectAddress$$

Operation: refunding a customer

Define pre $refund\_cust = preRefundCust$. The simplified schema is:

preRefundCust

$ISales$

$amount? : Money;\ cust? : Customer;\ date? : Date$

$cust? \in \text{ran}\ custaccounts$

Only one error may occur, that is, when the customer does not have an account with the company.

UnknownCust

$\Xi ISales$

$cust? : Customer;\ resp! : Report$

$cust? \notin \text{ran}\ custaccounts \land resp! = UnknownCustomer$

The total operation for this operation is therefore:

$$RefundCust \mathrel{\widehat=} refundCustOk \lor UnknownCust$$

Operation: Cash deposit

Define pre $ReceivCash = preReceivCash$. The simplified schema of this operation is:

preReceivCash

$Agency$

$cust? : Customer;\ amount? : Money;\ id? : Agencyid$

$date? : Date$

$id? \in agencies$

$\exists\, ag : Agency \bullet ag.identifier = id? \land cust? \in \text{ran}\ ag.ssales.accounts$

When this precondition schema is negated, a list of error conditions (for which operations were described in the previous cases) are obtained. The schema calculus expression of the

total operation is therefore:

$$ReceivCash \mathrel{\widehat=} receivCashOk \lor UnknownIdentifier \lor AgencyNotFound \lor UnknownCust$$

Operation: Allocating a transaction to an agency's account
The schema of the precondition for this operation is:

preAllocateTrans

$Agency$
$cust? : Customer;\ id? : Agencyid;\ datetrans? : Date$

$cust? \in \text{dom}(\text{dom}(\text{dom}(cashin \rhd id?)))$
$id? \in \text{ran}(agencyaccounts)$

As with the previous cases, the schema of the precondition is negated to yield error conditions. Having $cust? \notin \text{dom}(\text{dom}(\text{dom}(cashin \rhd id?)))$ implies that the customer has not issued any payment, and the operation to allocate the customer's transaction to the agency's account fails. The system then, continues with *TransNotFound* to handle the error.

TransNotFound

$\Xi Agency$
$cust? : Customer;\ resp! : Response$

$cust? \notin \text{dom}(\text{dom}(\text{dom}(cashin \rhd id?)))$
$resp! = TransactionNotFound$

Another error occurs when no account was created for the target agency.

AgencyAccountNotFound

$\Xi Agency$
$cust? : Customer;\ id? : Agencyid$
$resp! : Report$

$id? \notin \text{ran}(agencyaccounts) \land resp! = AgencyAccountNotFound$

The total operation is summarised as:

$$AllocateTrans \mathrel{\widehat=} allocateTransOk \lor TransNotFound \lor AgencyAccountNotFound$$

Operation: Notification of the other agency about a cash deposit
The schema for the precondition of this operation is shown below:

preNotifyCustTrans

$Agency;\ System$
$cust? : Customer;\ trans? : Transaction$
$datenotice? : Date;\ id? : Agencyid$

$id? \in \text{ran}(agencyaccounts)$
$cust? \in \text{dom}(\text{dom}(\text{dom}(cashin \rhd id?))) \land$
$trans? \in \text{ran}(\text{dom}(\text{dom}(statements)))$
$\exists\, ag : Agency \bullet ag.identifier = id? \land cust? \in \text{ran}(ag.ssales.custaccounts)$

The precondition *preNotifyCustTrans* is negated to yield error conditions. So, the first error to consider is when the system cannot determine a transaction made by a customer. That is, when $cust? \in \text{dom}(\text{dom}(\text{dom}(cashin \rhd id?)))$. This error was previously defined, and the operation *TransNotFound* was described to handle the error.
The next error occurs when a payment made at a cashier is not yet allocated to any agency's account.

TransNotAllocated

$\Xi Agency$
$cust? : Customer;\ trans? : Transaction;\ resp! : Report$

$cust? \in \text{dom}(\text{dom}(\text{dom}(cashin \rhd id?)))$
$trans? \notin \text{ran}(\text{dom}(\text{dom}(statements)))$
$resp! = TransactionNotAllocated$

At this point, all possible error conditions have been specified and will simply be re-used in further total operations when needed. Observe, that some of the operations so far described may need to be promoted to accommodate their application environment. As mentioned before, we are not considering such structuring activities here for the reasons stated in Section 3.4, P. 40.

UnknownCustRemote

$\Xi Agency;\ \Xi System$
$cust? : Customer;\ id? : Agencyid;\ resp! : Report$

$id? \in \text{ran}(agencyaccounts)$
$\exists\, ag : Agency \in known \bullet ag.identifier = id? \land cust? \notin \text{ran}(ag.ssales.custaccounts)$
$resp! = UnknownCustomer$

The total operation is:

$$notifyCustTrans \mathrel{\widehat{=}} NotifyCustTransOk \lor TransNotFound \lor AgencyNotFound \lor AgencyAccountNotFound \lor UnknowCustRemote$$

The table presented next, gives a summary of the total operations.

3.4.5 Table of total operations

Operation	Inputs and Outputs	Preconditions
ReceivItem	$item? : Item$ $inv? : Invoice$ $id? : Agencyid$ $date? : Date$ $resp! : Report$	$item? \notin \text{ran}(\text{dom}(\text{dom}\, itemsin))$ $id? \in agencies$ $\exists\, ag : Agency \bullet ag.identifier = id? \land$ $\quad inv? \in ag.ssales.invoices$
SendItem	$item? : Item$ $id? : Agencyid$ $dateout? : Date$ $addr? : Address$ $resp! : Report$ $dateallocated? : Date$	$item? \notin \text{dom}(\text{dom}\, itemsout)$ $item? \in \text{ran}(\text{dom}(\text{dom}\, itemsin))$ $id? \in \text{ran}\, itemsin$ $id? \in agencies$ $\exists\, ag : Agency \bullet ag.identifier = id? \land$ $\quad ag.addr = addr?$
RefundCust	$amount? : Money$ $cust? : Customer$ $date? : Date$ $resp! : Report$	$pers? \in custaccounts$
ReceivCash	$cust? : Customer$ $id? : Agencyid$ $amount? : Money$ $date? : Date$ $resp! : Report$	$id? \in agencies$ $\exists\, Agency \bullet \theta Agency.identifier = id? \land$ $\quad cust? \in \text{ran}(\theta Agency.ssales.accounts)$
AllocateTrans	$cust? : Customer$ $id? : Agencyid$ $datetrans? : Date$ $resp! : Report$	$cust? \in \text{dom}(\text{dom}(\text{dom}(cashin \rhd id?)))$ $id? \in \text{ran}(agencyaccounts)$
NotifyCustTrans	$cust? : Customer$ $id? : Agencyid$ $trans? : Transaction$ $datenotice? : Date$ $resp! : Report$	$id? \in \text{ran}(agencyaccounts)$ $cust? \in \text{dom}(\text{dom}(\text{dom}(cashin \rhd id?)))$ $trans? \in \text{ran}(\text{dom}(\text{dom}\, statements))$ $\exists\, Agency \bullet \theta Agency.identifier = id? \land$ $\quad cust? \in \text{ran}(\theta Agency.ssales.custaccounts)$

Table 3.1: Summary of total operations

Each of the three high-level services of the system are defined.
$DefineReturnitem \mathrel{\widehat{=}} ReceivItem \fatsemi SendItem \fatsemi RefundCust$

$DefineReplaceItem \mathrel{\widehat=} ReceivItem \fatsemi SendItem$
$DefinePaycredit \mathrel{\widehat=} ReceivCash \fatsemi AllocateTrans \fatsemi NotifyCustTrans$

3.5 Observations on Z

The following may be observed from the above Z specification:

1. The elegance of using Z to specify a system is highlighted. Each abstract state of the system is described in detail. For each operation, the precondition is calculated and the error conditions are clearly determined.

2. Abstract state spaces and operations are specified at the same level of abstraction as in the informal requirements definition.

3. Apart from using schema calculus operators to combine different schemas to form a new one, components from one schema, may be included or used in formulas within another schema.

Although these observations bring to light the precision that Z adds into the description of the functionalities of a system, they also demonstrate some limitations of the notation. E.g. they suggest the difficulties to modify, for example, an operation after the specification is built, since different components may be affected. They also support the idea of using a formal specification technique after some anlaysis and refinement of the initial requirements have been performed, as users generally do not initially know what they want until some pre-analysis of their needs is performed. Another important point, from the above observations, is that using Z at this stage, may result in specifying only the user-view of the system, instead of the system functionalities that may be obtained by refining the initial goals or user requirements.

3.6 Chapter summary

This chapter presented a natural language (English) description of the case study used in this work. A UCM model and a Z specification for the case study were suggested. Some of the advantages of using the UCM technique to capture and model user requirements were presented. Also a number of observations on the use of Z were put forward.

In the following chapter, based on the knowledge gained from the literature relating concepts in Z and Object-Z, and some guidelines proposed to help transform a Z document into Object-Z, the Z specification developed in this chapter, is transformed into Object-Z.

Chapter 4

Transforming the Z specification

Chapter 3 described the case study used in this work and proposed a UCM model and a Z specification for the case study. This chapter aims to transform the Z specification (see Chap. 3, Sec. 3.4) into Object-Z (see Figure 1.3). The following section presents the transformation process.

4.1 Transformation process

The feasibility of transforming a Z document into Object-Z, is supported by the fact that Object-Z is itself an extension of Z to accommodate Object-orientation (Carrington and Smith [22]). The general idea is, therefore, to group Z schemas to form Object-Z class schemas. To achieve this structuring work, some changes need to be made to those schemas, as well as the initial Z types and axiomatic definitions, to make them amenable to their new object-oriented environment. For example, a basic type which denotes some set of objects in Z, needs to be transformed into a class in Object-Z, to allow for the use of the same set of objects in Object-Z. Details regarding the rules of transformation applicable to each particular type of element(s) in Z, are not explicitly presented here however, Periyasamy and Mathew [68] provide important guidelines in this regard. During the transformation process, a summary of the mechanism used is given for each case. When necessary, detailed explanations are also provided whenever the idea may be unclear or not explicitly addressed in the referenced guide document.

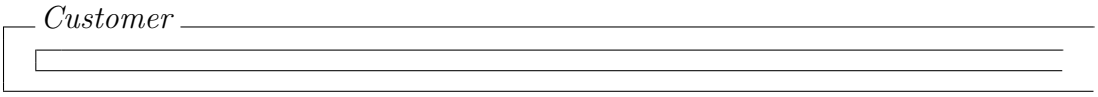
The next section describes on the step-by-step transformation.

4.2 The Object-Z specification

Two basic steps are followed to create the Object-Z version of the input specification. Firstly, all the Z basic types are transformed. And secondly, class schemas are created, based on the transformation of the initial Z axiomatic definitions, state schemas and operation schemas. Descriptive text attached to those elements are, when necessary, modified to accommodate the changes.

4.2.1 Basic types

Since a given type in Z represents a set of undefined objects, the Object-Z transformation of such a type may yield a class containing no state and no operation, namely, an empty class. For that reason, each given type of the input specification is therefore transformed into an empty class. The name of the class will be kept identical to that of the Z-type for re-usability purposes. For brevity, only one such class will be represented graphically to illustrate the idea, and the rest will be assumed. Below is the graphical representation of the class *Customer*; the Object-Z transformation of the given type *Customer*.



As stated above, an instance of this class is a customer object. At this stage nothing is said about the properties and behaviour of a customer. Nevertheless, the class can be used in any other class, for example, by instantiation (the new class refers to it, as an attribute of its variable), or by inheritance. The list of empty classes resulting from the transformation are:

Item	the class of items
Customer	the class of all possible customers
Agencyid	the class of class identifiers
Invoice	the class of all invoices
Currency	the class of all possible currencies
Transaction	the class of transactions
AccountNo	the class of account numbers
Money	the class all possible amounts of money
Language	the class of all human languages
Report	the class of all possible messages that may be exchanged with the system
Address	the class of all possible addresses
Date	the class of dates.

Note that Z basic types are part of Object-Z and may be incorporated directly into Object-Z classes. The difficulty would be to partition the initial list of types, in a way to place only those that are required into a class.

The Object-Z class schemas for the input Z specification are presented next.

4.2.2 Class schemas

The Object-Z classes resulting from the transformation of the input Z specification are generated from the following premises:

- For each abstract state space, a class schema is created. The name of the class is that of the state schema with "Cls" added to it as prefix. The unnamed version of the abstract state is added to the class as its only state.
- A realisable initial state of the state space from which a class is formed, is also added to the class.
- Any operation that applies to a state, from which a class is generated, is added to the class. The delta operator (Δ) in the definition of an operation within a class indicates the components that are changed by the operation. The Z operator Xi (Ξ) is simply eliminated from the operation.
- Any variable used in an operation is defined in the class containing the operation. Because global variables may be used in different operations and classes, and hence, be defined multiple times, this may cause confusion. For that reason, it is suggested a single class that encapsulates all the global variables be created, this allows each class to inherit variables from such a single class.

The next paragraph describes the class of global variables.

The class of global variables

This class encapsulates the global variables defined in the input Z specification. The meaning of each variable within the class remains unchanged from that of Chapter 3 (Section 3.4.1, p.41). The purpose in creating this class is to make those variables accessible to any other class, for example by inheritance.

ClsGlobalVariables

$\lceil(exchange, translate, amount, Report)$

$exchange : Money \times Currency \times Currency \pfun Money$

$translate : Message \times Language \times Language \pfun Message$

$amount : Transaction \pfun Money$

$Success,$
$ItemAlreadyReturned,$
$UnknownIdentifier$
$AgencyNotFound,$
$InvalidInvoice,$
$ItemAlreadySent,$
$ItemNotReceived,$
$IncorrectAddress,$
$TransactionNotFound,$
$UnknownCustomer,$
$AgencyAccountNotFound,$
$PaymentNotFound,$
$TransactionNotAllocated$

$\langle Success, ItemAlreadyReturned, UnknownIdentifier, AgencyNotFound,$
$InvalidInvoice, ItemAlreadySent, ItemNotReceived, IncorrectAddress,$
$TransactionNotFound, UnknownCustomer, AgencyAccountNotFound,$
$PaymentNotFound, TransactionNotAllocated\rangle \in iseqReport$

The class of "Accounts"

The class *ClsAccount* results from the transformation of the Z abstract state *Account* that describes the properties of an account object. The class includes only the operation of initialisation as a method, because in the input Z specification, there is no other operation that applies exclusively to the state *Account*. The components of the class are accessible and can be initialised from the environment of the system. Each object of the class includes a unique number and a balance which is initially equal to zero.

ClsAccount

$\lceil(accountno, balance, INIT)$

$accountno : AccountNo$
$balance : Money$

$account \neq \bot$

INIT

$balance' = 0 \land accountno' \neq \bot$

Next is the class of objects known as communication interfaces between this system and a local sales system in an agency.

The class of "Communication interfaces"

The class *ClsISales* results from the transformation of the Z abstract state *ISales*. It includes components that are provided by a local sales system, in an agency, to facilitate the communication with this system, as indicated in Figure 3.2, Chapter 3. The description of these components as given in Chapter 3, Section 3.4.2, on page 42 remains unchanged. It inherits the variables *Report*, and the function *amount* from the class of global variables. The variable *Message* is defined to promote the list of reports into the class, and *getAmount* is defined to render the function *amount*, from the class *ClsGlobalVariables*, usable in *ClsISales*.

ClsISales

$\upharpoonright(custaccounts, statements, invoices, \mathrm{INIT}, RefundCust)$

$$
\begin{array}{l}
var : ClsGlobalVariables \\
custaccounts : ClsAccount \rightarrowtail Customer \\
statements : ClsAccount \times Transaction \rightarrow\!\!\!\!\!+ Date \\
invoices : \mathbb{P}\, Invoice \\
\hline
\mathrm{dom}(\mathrm{dom}\, statements) \subseteq \mathrm{dom}\, custaccounts
\end{array}
$$

INIT

$$
custaccounts = \varnothing \wedge statements = \varnothing \wedge invoices = \varnothing
$$

$$
Message \mathrel{\widehat=} var.Report \;\; \wedge \;\; getAmount \mathrel{\widehat=} var.amount
$$

refundCustOk

$$
\begin{array}{l}
\Delta(statements) \\
amount? : Money;\; cust? : Customer;\; date? : Date \\
resp! : Message \\
\hline
\exists\, trans : Transaction \bullet \\
\quad trans \notin \mathrm{ran}(\mathrm{dom}\, statements) \wedge getAmount(trans) = amount? \\
\quad \exists\, Account \bullet \\
\qquad \theta Account \mapsto cust? \in custaccounts \\
\qquad statements' = statements \cup (\theta Account, trans) \mapsto date? \\
resp! = Success
\end{array}
$$

UnknownCust

$$
\begin{array}{l}
cust? : Customer;\; resp! : Report \\
\hline
cust? \notin \mathrm{ran}\, custaccounts \wedge resp! = UnknownCustomer
\end{array}
$$

$$
RefundCust \mathrel{\widehat=} RefundCustOk \;[\!]\; UnknownCust
$$

The class defines the total operation *RefundCust* (as described in the previous chapter) and makes it visible to other classes. Components of the class are made accessible to other classes, except for the variable *var*.

The following class describes database components required in each agency.

The class of "Databases"

The class *ClsDatabase* is the Object-Z transformation of the Z abstract state space *Database*. Similar to the class *ClsAccounts*, the only method of the class is the operation of initialisation. It includes the Z state schema, namely, *database* and its initial state, as defined in Chapter 3. All the components in the state are made accessible to other classes and can be initialised from the environment.

ClsDatabase

$\upharpoonright(agencies, agencyaccounts, itemsin, itemsout, collected, cashin, statements, \mathit{INIT})$

$$
\begin{array}{l}
agencies : \mathbb{P}\, Agencyid \\
agencyaccounts : ClsAccount \rightarrowtail\!\!\!\!\!\!\leftrightarrow Agencyid \\
itemsin : Invoice \times Item \times Date \twoheadrightarrow\!\!\!\!\!\!\!\!\mapsto Agencyid \\
itemsout : Item \times Date \nrightarrow Address \\
collected : Item \times Agencyid \nrightarrow Date \\
cashin : Customer \times Money \times Date \nrightarrow Agencyid \\
statements : ClsAccount \times Transaction \nrightarrow Date \\
\hline
\mathrm{ran}(agncyAccounts) \subseteq agencies \wedge \mathrm{ran}(itemsin) \subseteq agencies \\
\mathrm{dom}(\mathrm{dom}\, itemsout) \subseteq \mathrm{ran}(\mathrm{dom}(\mathrm{dom}\, itemsin)) \\
\mathrm{ran}\, cashin \subseteq \mathrm{ran}\, agencyaccounts \\
\mathrm{dom}(\mathrm{dom}\, statements) \subseteq \mathrm{dom}(agencyaccounts) \\
\mathrm{ran}(\mathrm{dom}\, collected) \subseteq agencies \\
(\forall Account \mid \theta Account \in \mathrm{dom}(statements)) \bullet \\
\quad (\exists id : Agencyid \bullet id \in \mathrm{ran}(cashin) \wedge \theta Account \mapsto id \in agencyaccounts)
\end{array}
$$

InitDatabase

$$
\begin{array}{l}
Agencies' = \varnothing \wedge agencyaccounts' = \varnothing \wedge itemsin' = \varnothing \\
itemsout' = \varnothing \wedge cashin' = \varnothing \wedge statements' = \varnothing \wedge collected' = \varnothing
\end{array}
$$

This class is meant to be used by other classes; notably by the class of agencies presented next.

The class of "Agencies"

The class *ClsAgency*, is the Object-Z transformation of the Z state schema *Agency* from the input Z specification. An object of this class inherits properties from the two classes

ClsDatabase and *ClsISales* as a result of having the two abstract states spaces, *Databases* and *ISales* included in the state space *Agency*, from the input Z specification. The component *db* defines the identity of an object of the class of databases; this object encapsulates the data used locally in an agency. The component *ssales* defines the identity of an object of the class *ClsISales* to handle communications between an object of this class, representing an agency, with other agencies. Global variables are inherited from the class *ClsGlobalVariables*.

ClsAgency

$$\lceil (db, ssales, identifier, dcurrency, address, language, ReceivItem, ReceivCash, AllocateTrans)$$

ClsGlobalVariables

$$db : ClsDatabase$$
$$ssales : ClsISales$$
$$identifier : Agencyid$$
$$dcurrency : Currency$$
$$address : Address$$
$$language : Language$$

$$identifier \notin agencies$$
$$\mathrm{dom}(ssales.custaccounts) \cap \mathrm{dom}(agencyaccounts) = \varnothing$$

INIT

$$db.INIT \land ssales'.custaccounts = \varnothing$$
$$ssales'.statements = \varnothing \land ssales'.invoices' = \varnothing$$

receiveItemOk

$$\Delta(db.itemsin)$$
$$item? : Item;\ inv? : Invoice;\ id? : Agencyid;\ date? : Date$$
$$addr! : Address;\ lang! : Language;\ resp! : Report$$

$$item? \notin \mathrm{ran}(\mathrm{dom}(\mathrm{dom}\ itemsin)) \land id? \in agencies$$
$$(\exists\, Agency \bullet$$
$$\quad \theta Agency.identifier = id? \land inv? \in \theta Agency.ssales.invoices \land$$
$$\quad addr! = \theta Agency.address \land lang! = \theta Agency.language)$$
$$db.itemsin' = db.itemsin \cup (inv?, item?, date?) \mapsto id? \land resp! = Success$$

ItemAlreadyReturned

$$item? : Item;\ resp! : Report$$

$$item? \in \mathrm{ran}(\mathrm{dom}(\mathrm{dom}\ db.itemsin)) \land resp! = ItemAlreadyReturned$$

UnknownIdentifier

$$id? : Agencyid;\ addr! : Address$$

$$id? \notin db.agencies \land resp! = UnknownIdentifier$$

AgencyNotFound

$id? : Agencyid;\ resp! : Report$

$id? \in db.agencies \land \forall\, ag : Agency \bullet ag.identifier \neq id?$
$resp! = AgencyNotFound$

InvalidInvoice

$inv? : Invoice;\ id? : Agencyid$
$resp! : Report$

$id? \in agencies \land$
$\exists\, ag : Agency \bullet ag.identifier = id? \land inv? \notin ag.ssales.invoices$
$resp! = InvalidInvoice$

$$ReceivItem \mathrel{\widehat=} ReceivItemOk \,[]\, ItemAlreadyReturn \,[]\, UnknownIdentifier \,[]\, AgencyNotFound \,[]\, InvalidInvoice$$

receivCashOk

$\Delta(db.cashin)$
$cust? : Customer;\ amount? : Money;\ id? : Agencyid;\ date? : Date$
$resp! : Report$

$\exists\, agency : ClsAgency \mid agency.identifier = id? \bullet$
$\quad \exists\, account : ClsAccount \bullet accoung \mapsto cust? \in agency.ssales.accounts$
$db.cashin' = db.cashin \cup \{(cust?, amount?, date?) \mapsto id?\}$
$resp! = Success$

$$ReceivCash \mathrel{\widehat=} ReceivCashOk \,[]\, UnknownIdentifier \,[]\, AgencyNotFound \,[]\, UnknownCust$$

allocateTransOk

$\Delta(db.statements)$
$cust? : Customer;\ id? : Agencyid;\ date? : Date$
$resp! : Response$

$(\exists\, account : ClsAccount;\ trans : Transaction;\ date : Date) \bullet$
$\quad (cust?, amount(trans), date) \mapsto id? \in cashin \land$
$\quad account \mapsto id? \in agencyaccounts$
$db.statements' = db.statements \cup (account, trans) \mapsto date?$
$resp! = Success$

TransNotFound

$cust? : Customer;\ resp! : Response$

$cust? \notin \mathrm{dom}(\mathrm{dom}(\mathrm{dom}(db.cashin \rhd id?)))$
$resp! = TransactionNotFound$

AgencyAccountNotFound

$cust? : Customer;\ id? : Agencyid$

$resp! : Report$

$id? \notin \text{ran}\ db.agencyaccounts \land resp! = AgencyAccountNotFound$

$AllocateTrans \mathrel{\widehat=} AllocateTransOk \ [] \ TransNotFound \ [] \ AgencyAccountNotFound$

The operations of the class are: the initialisation operation and three other operations.

1. *ReceivItem*: An object of the class representing an agency uses this operation to handle the receipt of an item returned by a customer that is to be forwarded to another agency.

2. *ReceivCash*: This operation is used within an agency to receive cash deposited by a customer for a credit payment, to the benefit of another company.

3. *AllocateTrans*: Transfer of cash deposited by a customer into the appropriate account.

Each of these operations is composed of an operation that describes the behaviour of the system under normal circumstances, and a set of other operations defining error conditions as required by standard Z. Note that in Object-Z, the idea of calculating precondition is not applicable, since each operation is independently defined without being subjected to be either total, partial or meant to complement another operation by specifically handling errors. In this regard, using Z as an intermediary step in an Object-Z specification, is advantageous in benefiting from the precondition calculation, which helps to evaluate different conditions that may in turn influence the execution of an operation. Each of the composed operations becomes applicable when the precondition of one of the composing operations is satisfied. In that case, the operation, for which the precondition is satisfied, is internally selected by the system.

All the components of an object of the class are accessible to its environment from which the components may be initialised. The three main operations of the class are made available from outside the class.

Next the class schema that describes the entire system is presented .

The class System

The class *ClsSystem* of objects, representing a copy of the system as perceived from an agency, is derived from the state schema *System* of the input Z specification. The component

this is added to reference the agency in which the system is operating and distinguishes it from the others. Operations locally performed in an agency, are thus accessible through *this*.

ClsSystem

$\lceil(\mathit{INIT}, \mathit{SendItem}, \mathit{NotifyCustTrans})$

$\mathit{this} : \mathit{ClsAgency}; \qquad \mathit{known} : \mathbb{P}\, \mathit{ClsAgency}$

$\mathit{this} \notin \mathit{known}$

INIT

$\mathit{this}.\mathit{INIT} \land \mathit{known} = \varnothing$

sendItemOk

$\Delta(\mathit{this}, \mathit{known})$
$\mathit{item}? : \mathit{Item};\ \mathit{dateout}?, \mathit{datecollected}? : \mathit{Date}$
$\mathit{addr}? : \mathit{Address};\ \mathit{id}? : \mathit{Agencyid};\ \mathit{resp}! : \mathit{Report}$

$\mathit{item}? \in \mathrm{ran}(\mathrm{dom}(\mathrm{dom}\ \mathit{this.db.itemsin}))$
$(\exists\, \mathit{inv} : \mathit{Invoice};\ \mathit{datein} : \mathit{Date}) \bullet$
$\quad (\mathit{inv}, \mathit{item}?, \mathit{datein}) \mapsto \mathit{id}? \in \mathit{this.db.itemsin}$
$\quad \mathit{this.db.itemsout}' = \mathit{this.db.itemsout} \cup (\mathit{item}?, \mathit{dateout}?) \mapsto \mathit{addr}?$
$(\exists\, \mathit{agency} : \mathit{ClsAgency} \mid \mathit{agency} \in \mathit{known}) \bullet$
$\quad \mathit{agency.identifier} = \mathit{id}? \land \mathit{agency.addr} = \mathit{addr}?$
$\quad \mathit{agency.db.collected}' = \mathit{agency.db.collected} \cup$
$\qquad (\mathit{item}?, \mathit{id}?) \mapsto \mathit{datecollected}?$
$\mathit{resp}! = \mathit{Success}$

ItemAlreadySent

$\mathit{item}? : \mathit{Item};\ \mathit{resp}! : \mathit{Report}$

$\mathit{item}? \in \mathrm{ran}(\mathrm{dom}(\mathrm{dom}\ \mathit{this.db.itemsin})) \land \mathit{item}? \in \mathrm{dom}(\mathrm{dom}\ \mathit{this.db.itemsout})$
$\mathit{resp}! = \mathit{ItemAlreadySent}$

ItemNotReceiv

$\mathit{item}? : \mathit{Item};\ \mathit{resp}! : \mathit{Report}$

$\mathit{item}? \notin \mathrm{ran}(\mathrm{dom}(\mathrm{dom}\ \mathit{itemsin})) \land \mathit{resp}! = \mathit{ItemNotReceived}$

IncorrectAddress

$\mathit{addr}? : \mathit{Address};\ \mathit{id}? : \mathit{Agencyid}$
$\mathit{resp}! : \mathit{Report}$

$\mathit{id}? \in \mathit{agencies}$
$\exists\, \mathit{agency} : \mathit{ClsAgency} \bullet \mathit{agency.identifier} = \mathit{id}? \land \mathit{agency.addr} \neq \mathit{addr}?$
$\mathit{resp}! = \mathit{IncorrectAddress}$

$\mathit{SendItem} \mathrel{\widehat=} \mathit{SendItemOk} \,[]\, \mathit{ItemAlreadySent} \,[]\, \mathit{ItemNotReceiv} \,[]$
$\qquad \mathit{AgencyNotFound} \,[]\, \mathit{IncorrectAddress}$

notifyCustTransOk

$\Delta(known)$
$cust? : Customer;\ trans? : Transaction;\ datenotice? : Date;\ id? : Agencyid$
$resp! : Report$

$(\exists\, agency : ClsAgency \mid agency \in known \land$
$\exists\, account, custaccount : ClsAccount;\ date1, date2 : Date) \bullet$
$(cust?, amount(trans?), date1) \mapsto id \in this.db.cashin \land$
$(account, trans?) \mapsto date2 \in this.db.statements$
$agency.identifier = id? \land account \mapsto id? \in this.db.agencyaccounts$
$custaccount \mapsto cust? \in agency.ssales.custaccounts$
$agency.ssales.statements' =$
$(agency.ssales.statements \cup (custaccount, trans?) \mapsto datenotice)$
$resp! = Success$

TransNotAllocated

$cust? : Customer;\ trans? : Transaction;\ resp! : Report$

$cust? \in \mathrm{dom}(\mathrm{dom}(\mathrm{dom}(this.db.cashin \rhd id?)))$
$trans? \notin \mathrm{ran}(\mathrm{dom}(\mathrm{dom}(this.db.statements)))$
$resp! = TransactionNotAllocated$

UnknownCustRemote

$cust? : Customer;\ id? : Agencyid;\ resp! : Report$

$id? \in \mathrm{ran}(this.db.agencyaccounts)$
$\exists\, ag : Agency \in known \bullet ag.identifier = id? \land cust? \notin \mathrm{ran}(ag.ssales.custaccounts)$
$resp! = UnknowCustomer$

$NotifyCustTrans \mathrel{\widehat{=}} NotifyCustTransOk \,[]\, TransNotAllocated \,[]\, this.AgencyNotFound$
$[]\, this.AgencyAccountNotFound \,[]\, UnknowCustRemote$

An object of this class defines two composed operations that involve at least two agencies. Those are:

1. *SendItem*, to forward an item returned by a customer to the appropriate agency where the item was purchased.

2. *NotifyCustTrans*, to update the account of a customer in the agency, in which the customer has deposited some amount of money.

These operations are accessible from the environment of the system. The user can also initialise the components of the system.

A brief summary of the chapter follows.

4.3 Chapter summary

This chapter concentrated on transforming the Z specification of the case study, (presented in Chapter 3) into Object-Z. The transformation process was guided by knowledge from the literature, and supported with concepts from ordinary Z, Object-Z and Object-orientation.

As observed earlier, a remarkable advantage of this transformation is to exploit the Z precondition calculation to determine error conditions for the operations before integrating them into Object-Z classes.

The next chapter presents a framework for transforming a Use Case Map (UCM) model of a system into Object-Z.

Chapter 5

A Framework for transforming a UCM into Z and Object-Z

In Chapter 3 the description of the case study was presented, and a UCM map was constructed from it. The purpose of this chapter is to demonstrate the transformability of a UCM to a Z and an Object-Z specification, by proposing a framework to translate generic elements of UCMs to Z and Object-Z elements. Examples are given to illustrate concepts that may not be easily understandable. The basic transformation strategy, supported by a small diagram, is first presented in Section 5.1. This is followed in Section 5.2 with the analysis of the threefold relationship between UCMs, Z and Object-Z, as well as the analysis, in Section 5.3, of concepts used in UCMs, Z and Object-Z. Thereafter, the proposed transformation process is presented in Section 5.4, prior to a conclusion in Section 5.5.

The main concept in this chapter constitutes one of the important contributions of this dissertation. The summary was presented, as a research paper, at the 7th International Workshop on Modelling Simulation Verification and Validation of Enterprise Information Systems (MSVVEIS 2009) in Milan Italy (Dongmo and van der Poll [23]).

5.1 Basic transformation strategy

Although UCM, as a semi-formal notation, may share with natural languages some limitations, such as allowing ambiguous requirements, non-detection of errors, etc., it has the advantage of encapsulating different types of information in a single view. Thus, a drawback of a transformation process, would be the loss of information (e.g. when a UCM is transformed into a Message Sequence Chart, some information on the scenario interactions is lost (Miga et al. [60])). The architecture of the proposed mechanism is presented in Figure 5.1. Z is used as an intermediate transformation step. This way, the rigour of Z may be exploited to

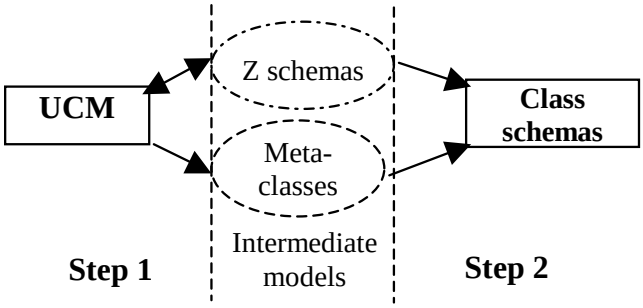


Figure 5.1: Basic transformation strategy

allow for clear and precise definitions of static and dynamic behaviours of systems, extracted from an input UCM. At the same time, meta-classes are used to extract necessary architectural information. Thereafter Z schemas are combined with the meta-classes, to form the Object-Z class schemas. A two-step transformation mechanism is presented in more detail in Section 5.4.

In the diagram above, the double-headed arrow between the input UCM, and the Z schemas, indicates possible iteration at the level of unit component. That is, after a conceptual unit element of UCM is transformed, the precision brought into the corresponding Z schema(s) may help, reverse-wise to improve the initial UCM element. For example, the precondition of a critical operation may be formally calculated in Z, and the result reported back to improve its description in UCM.

At the level of the system, the double-headed arrow indicates the fact that the corresponding Z description may be simply used to improve the input UCM and hence, serve as a through away specification, to suit the need of a designer who prefers, for example, to work with graphical models, since these may be easier to manipulate, and enhance communication between stakeholders.

5.2 Relationship between UCM, Z and Object-Z

To evaluate the feasibility of the above mechanism, it is necessary to analyse the relationships between the UCM, Z and Object-Z notations:

- Both notations are specification techniques that focus on system functionalities at the requirement level; both can also be used during later stages of the software development process.

- Their documentations includes, for clarification purposes, natural language descriptions aimed at explaining the possible intricacies of UCMs and schemas.

- UCMs target the static, dynamic and architectural aspects of a system, while Z focuses on the static and dynamic aspects only. However, the architectural component of UCMs can be compensated with the class structures of Object-Z (see Figure 5.1).

- Users and industries may more easily adopt the usage of UCMs. This might be because the UCM notation is graphic in nature, and therefore more appealing to humans than the terse mathematical notation of Z. Formal methods tend to be perceived by industry as being unsuitable for serious system design. It is believed that this situation stems from the fact that the Established Strategy for constructing Z documents (van der Poll and Kotzé [93], van der Poll et al. [94]), is largely silent about the architecture of a system. Schemas are defined, and it is left to the user to perceive how these fit together in the final system. Some suggestions, notably principles to guide the construction process have been made (van der Poll and Kotzé [93], van der Poll et al. [94]); the difficulties seem to persist among practitioners since the said heuristics are still surrounded by technical terms. This is a further justification for using UCMs as an initial step in the use of a formal method.

- UCMs use scenario-based reasoning to target the general aspects of system functionality and structure, and are not concerned with detailed descriptions. Z, on the other hand, fills this gap as far as system functionality is concerned, but also fails to provide any construction process for the schemas. Sommerville (Sommerville [82]) suggests that formal methods in general, should be used at the system-requirements level, after the user requirements specification, but before any detailed design. This suggests that a one-to-one relationship between the elements of a UCM model and Z schemas may not be feasible in general, but the UCM elements may constitute important starting points in the construction of schemas.

Next, this analysis continues by investigating how concepts in UCM may be linked to those in Z and Object-Z.

5.3 Conceptualisations in UCM, Z, Object-Z

Since in a UCM, the two important concepts are paths (including path elements to describe scenarios) and components (to describe the architecture of the system), it is necessary to analyse the 3-tiered relationship between the above concepts in UCMs, schemas and types in Z and class schemas in Object-Z.

UCM path

A UCM path consists of one or more path segments. Each path segment includes, amongst others, path elements and a sequence of responsibilities, each representing an abstraction of service provided by the system. A path segment may be bound to a component that handles the execution of the responsibilities on such path. These UCM constructs may be modelled in Z, by a set of operation schemas to describe the bound responsibilities, a set of state schemas used to describe the portion of the system state that is controlled by the component and is likely to be consulted or changed by the bound responsibilities, and a list of basic types necessary to define the two sets of schemas mentioned above.

Responsibility points

A sequence of responsibility points in a path segment can, therefore, be modeled in Z, by schema composition. Alternatively, a Z sequence structure with schema operations as elements could be considered. A sequence structure may assist a specifier with traceability aspects of the transformed model.

Scenario interactions

Scenario interactions are represented in a UCM with path connectors, i.e. AND-fork, AND-join, OR-fork and OR-join. Such connectors may be described in Z using appropriate schema operators. Although the splitting and joining of path segments with those connectors may sometimes be associated together, in many real situations this might not be the case. That is, for instance, a UCM may include one or more OR-forks without any associated OR-join, and vice versa. For this reason, they are considered separately.

(a) OR-fork

Consider an OR-fork connector with one entry path segment and two outgoing alternatives (see Figure 5.2). Let Op1 be the composed schema that models the sequence of responsibilities of the entry path segment, and Op11 and Op12 schemas that specify the two alternative exit segments.

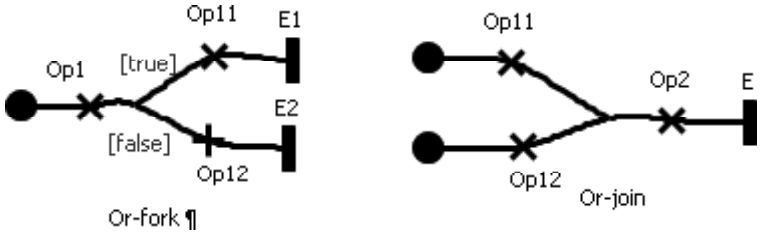


Figure 5.2: OR-fork and OR-join connectors

The resulting operation along such a path can therefore be described by the following Z schema calculus expression:

$$op1 \fatsemi (Op11 \lor Op12) \tag{5.1}$$

A text description may be added to indicate the selection policy at the OR-fork connection point, represented in the formula by the schema disjunction operator ($\lor$).

(b) OR-join

With this connector, the responsibilities along one incoming path segment are performed. This is followed sequentially by the responsibilities along the outgoing path segment. Hence, as in the case of the OR-fork, the activities around such a connector are described with the following expression:

$$(Op11 \lor Op12) \fatsemi Op2 \tag{5.2}$$

From both expressions, it may be inferred that the combination of OR-fork and OR-join connectors, could be described in Z, by the schema expression 5.3 below:

$$Op1 \fatsemi (Op11 \lor Op12) \fatsemi Op2 \tag{5.3}$$

Op1 represents a schema operation, or a schema expression, for the operations performed on the incoming path segment. Op11 is a schema expression for the operations on one of the two alternative path segments, Op12, the schema expression that describes the operations on the other alternative path segment, and Op2 the schema expression that describes the operations on the joined path segment.

When more than two alternative path segments are to be considered, more schema disjunction operators may be used to include, in the middle part of expression (5.3), operations performed on the additional path segments.

(c) AND-fork

Both the AND-fork and AND-join, are illustrated in Figure 5.3. Op1 represents the composed schema that models the sequence of responsibilities of the incoming path segment. And Op11 and Op12, respectively, represent the composed sequence of responsibilities on each of the synchronised path segments. The resulting operation along such a path in Z, is described with the following Z schema calculus expression:

$$Op1 \fatsemi (Op11 \land Op12) \tag{5.4}$$

Where $Op1$, $Op11$ or $Op12$ are, respectively, the schema expressions that describe Op1, Op11, and Op12. Text may be added to the schema expression in Formula 5.4 to provide more clarification on the synchronization policy.

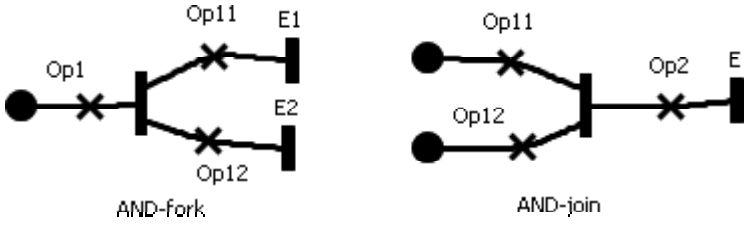


Figure 5.3: AND-fork and AND-join connectors

(d) And-join

With this connector, all the responsibilities along both incoming path segments are performed before any other responsibility along the outgoing path segment. Hence, if, as mentioned above, Op11, Op12, Op2 , describe operation along path segments (as in Figure 5.3), then the following schema calculus expression may describe the set of activities around the connector:

$$(op11 \wedge op12) \fatsemi op2 \tag{5.5}$$

In a situation where the AND-join is associated to an AND-Fork, the Z schema expression in Formula 5.6 may be appropriate to model the situation. And when multiple branches are synchronized, we may simply use more schema conjunction operators in the sub-expressions in the middle of Formula 5.6, to add the description of the operations on the additional branches.

$$Op1 \fatsemi (op11 \wedge op12) \fatsemi op2 \tag{5.6}$$

Waiting place

A waiting place is a notational element that implicitly blocks the execution of a scenario, progressing along a path (main path), waiting for some action along the clearing path to occur. In Z, such element may be described by defining a state schema to handle the state of the waiting place, some operation schemas to describe the above mentioned control activities, and a schema expression to combine them. For example, consider in Figure 5.4 the responsibility point "Rin" executed before entering the waiting place, "Rout" the one executed just after the main path is released, and "Cin", executed on the clearing path to unblock the main path. In Z, we may define an abstract state space, and two operation schemas, namely "block" and "release" to describe the situation.

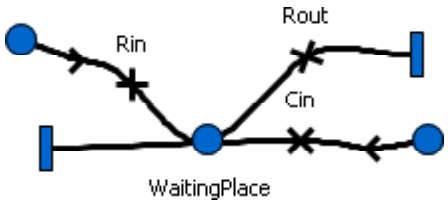


Figure 5.4: Waiting place

We define

$$[Identifier]$$

Identifier defines the set of all possible identification numbers that can be used to synchronise a blocking and release of a sequence of activities.

waitingPlace

$$opened : Boolean$$
$$waitingId : Identifier$$

$$opened = true \Rightarrow waitingid = \bot$$

When the system is not waiting for any clearing activity along the clearing path, the progression of any scenario may freely pass across the waiting place and the value of the variable *waitingId* is meaningless. This is to accommodate the fact that any path traversing the waiting place, may serve either as a main path or a clearing path.

block

$$\Delta WaitingPlace$$
$$id! : Identifier$$

$$opened = true$$
$$\exists\, id : Identifier \bullet waitingid = id \land id! = id$$
$$opened = false$$

This operation is executed immediately after the blocking responsibility "Rin" is performed. It generates an identifier that is passed to the clearing operation "Cin" to release the main path.

release

$$\Delta WaitingPlace$$
$$id? : Identifier$$

$$waitingId = id? \Rightarrow opened = true \land waitingId = \bot$$

This operation is triggered immediately after "Cin" is performed to unblock the execution of the scenario on the main path. The identifier passed to it by the clearing operation must

be identical to the one kept in the state variable *waitingId*. To model the complete sequence of activities along a waiting place, it is assumed *schemaRin*, *schemaRout*, and *schemaCin* are respectively the Z schemas, associated with the responsibilities "Rin", Rout, and "Cin", and then propose the Z schema calculus expression below:

$$(schemaRin \wedge block) \fatsemi (schemaCin \wedge release) \fatsemi schemaRout \tag{5.7}$$

The above model, presents a case where each time a waiting place is waiting to be released, all other instances of the blocking activities are also blocked. For example, consider Rin is an operation that sends information to remotely update a database, via a network, and waits for a response. The above model allows the next request to update the database to be forwarded only after the previous request has been responded to.

Timer

A Timer is a special case of a waiting place where the waiting time is limited. When a timeout occurs, an appropriate action (indicated by "Tout" in Figure 5.5) may be taken on the timeout path. To model a timer in Z, it is suggested to add one more variable to the above state space of a normal waiting place, and two more schema operations to handle the setting-up of the timer, and to take proper corrective action when a timeout occurs.

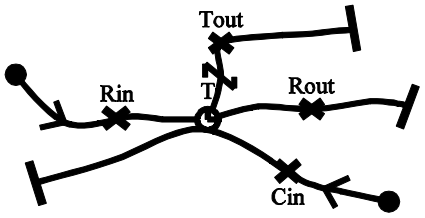


Figure 5.5: Timer

Timer

$$opened : Boolean$$
$$waitingId : Identifier;\ maxtime : \mathbb{N}$$

$$opened = true \Rightarrow waitingId = \bot \wedge maxtime = 0$$

The variable *maxtime* holds the maximum time allowed for a clearing action to occur. Otherwise, a timeout responsibility may be performed when such time expires.

setup

$\Delta Timer$

$time? : \mathbb{N}$

$time? > 0 \land maxtime = time?$

In this model, it is assumed that the operation that blocks the timer may decide on how long it will wait for a response from a clearing path, before taking any corrective action. Hence, the operation "BlockTimer" would be described as:

$$BlockTimer = block \land setup \tag{5.8}$$

where the operation *block* is defined as in the case of a waiting place, but operates on the state of *Timer*. The schema calculus expression to define the behaviour of the timer would be as in Formula 5.9 below.

$$(schemaRin \land blockTimer) \fatsemi ((schemaCin \lor schemaTout) \land release) \fatsemi schemaRout \tag{5.9}$$

with the operation *release* modelled as follows:

release

$\Delta Timer$

$id? : Identifier$

$waitingId = id? \Rightarrow$

$opened = true \land waitingId = \bot \land maxtime = 0$

Stubs

A stubbing technique is a mechanism used in Use Case Maps to defer some details of a map to sub-maps called plug-ins. There are two types of stubs: static and dynamic stubs. These will be considered separately.

(a) Static stub

with a static stub, the start points and end points of the bound plug-in are statically associated to the incoming and outgoing path segments of the stub. That is, whenever the execution gets into the stub through an incoming path, the same plug-in is reached, and as soon as the plug-in is executed, the progression continues through the outgoing path segments of the stub. As the sub-map is itself a complete map, it may therefore, be transformed separately. The challenge is to model in Z/Object-Z the integration of the plug-in into the main map. In this regard, the following is suggested:

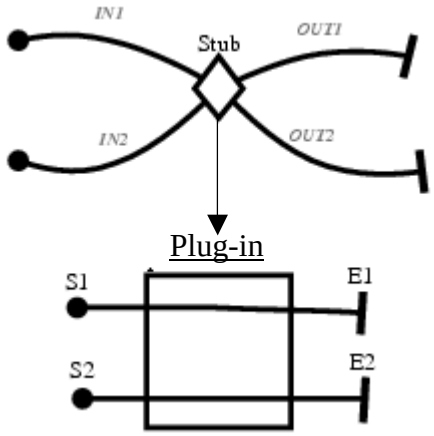


Figure 5.6: Example of a static stub

Define a list of basic types:

$$[Input, Output, Start, End]$$

where *Input* is the list of all possible path segments that enter into a stub. *Output* is the set of all path segments that goes out from a stub. Similarly, *Start* and *End* are the list of all possible start and end points of plug-ins.

Define a function that associates each input-path segment to a start-point.

$$Call : Input \pfun Start$$

Define a function that associates each end-point of the plug-in with an output of the stub that is, the list of static returning points after the plug-in has been executed.

$$Return : End \pfun Output$$

Based on the above, the state of a static stub can be defined as:

staticStub

$$callpoints : Input \pfun Start$$
$$returnpoints : End \pfun Output$$

$$predicate$$

It is assumed that each input from a stub is associated to only one start-point on the plug-in, and that such associations are not infinite. The component *callpoints* is hence, of a limited size and represents different possible ways the main maps may be linked to

the plug-in depending on the inputs. In the Object-Oriented concept, this would suggest polymorphism, where the behaviour of the plug-in depends on *callpoints* that link inputs to the appropriate arguments. Similarly, it is assumed that each end-point is mapped to one output, and the list of such mapping is of a limited size. The variable *returnpoints* represents such a list.

In the *predicate* part, depending on each particular case under consideration, appropriate formulas may be added to express possible constraints on inputs and outputs to indicate, for example, which outputs can be reached from given inputs. Descriptive text may also be included for more clarity.

At this point, two descriptive operations, *call* and *return*, may be defined respectively at runtime, to direct the execution of a scenario to the plug-in, and to return the execution back to the main map.

call

$$\Xi staticStub$$
$$in? : Input;\ s! : Start$$
$$in? \mapsto s! \in callpoints$$

In this model, to facilitate the understanding of our idea, a simple case, is considered when the execution reaches the stub from only one incoming path segment. But, in practice, multiple inputs may be involved concurrently; in this case, the list of those path segments, may be considered as the input to the function. Hence, the task of the function would be to find those start points, from where the execution will continue.
Next, the operation *return* is presented.

return

$$\Xi staticStub$$
$$end? : End;\ out! : Output$$
$$end? \mapsto out! \in returnpoints$$

(b) Dynamic stub

As pointed out by Amyot [6], dynamic stubbing is an interesting construct for scenario integration, as it may include multiple plug-ins, with only one of them being dynamically selected at runtime according to a selection policy. The transformation challenge for such constructs is, therefore, to describe in Z, the dynamic mapping of plug-ins. An example of a dynamic stub is presented in Figure 5.7. For the transformation purpose,

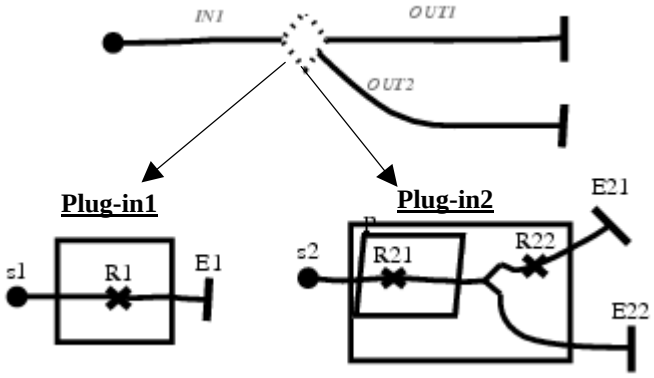


Figure 5.7: Example of a dynamic stub

we reconsider the basic types defined above for static stubbing, and propose the state schema *dynamicStub* described next.

dynamicStub

$$listInputs : \mathbb{F}\ Input$$
$$listStartPoints : \mathbb{F}\ Start$$
$$listEndPoints : \mathbb{F}\ End$$
$$listOutputs : \mathbb{F}\ Output$$

$$predicate$$

The variable *listInputs*, is the set of incoming path segments, to which Start-points from the set *listStartPoints* of plug-ins, may be connected. Similarly, *listEndPoints* is the set of outgoing path segments (on the stub), to which End-points from the set *listEndPoints* of plug-ins, may be connected.

As stated in the case of the static stub, depending on the particular dynamic stub under consideration, constraints on the state variables may be included in the *predicate* part. Descriptive prose text may also be added to facilitate the understanding of the model.

The implicit control operation of a dynamic stud is described next:

call

$$\Xi dynamicStub$$
$$in? : Input;\ s! : Start$$

$$\exists\, start : Start \bullet in? \mapsto start \in Input \pfun Start$$
$$s! = start$$

During runtime, when the execution of a scenario enters the stub through an input segment (e.g. $IN1$ in Figure 5.6), the system determines, according to a predefined selection

policy, the start point(s) of the plug-in that is to be executed. After the execution of the selected plug-in, the control (of the system) is returned, by linking the end point(s) of the plug-in to the appropriate output(s) of the stubs.

A Z model for the descriptive function *return* in the case of a dynamic stub is presented.

return

$$\Xi staticStub$$
$$end? : End;\ out! : Output$$

$$\exists\, out : Output \bullet end? \mapsto out \in End \nrightarrow Output$$
$$out! = out$$

The execution of the selected plug-in terminates at the end-point *end*?, where the system determines the appropriate output segment from which the execution will continue. This segment is returned as output. The selection of such an output may be described by means of predicates, or conditional statements, as they are mapped to end-points, only during runtime.

As the reader may notice, this chapter only considers those UCM path elements that were involved in the specification presented in Chapter 3. Modelling other UCM path elements is beyond the scope of this dissertation.

Active components

Active components, e.g. "Teams" and "Processes", have the responsibility to control the execution of responsibilities points bound to them (see Figure 5.8). Therefore, it is suggested that for each active component, an implicit generic operation (shared by all paths bound to the component), to control the execution of responsibilities, be considered, to this end, an additional schema operation should be created in Z to describe the generic control operation for each active component. Such "control" operations are traditionally not part of a Z specification. To illustrate the idea of a control operation that may be performed by active components, consider the example of UCM in Figure 5.8. The motivation in using this example is firstly, to make it clear that having a process in a map without any responsibility point bound to it is legal in UCM, and secondly, to show that in the absence of responsibility points bound to an active component, the component still have a purpose. For example, the component may be placed on the map to co-ordinate interaction between the scenario or their synchronization through path segments connectors bound to it.

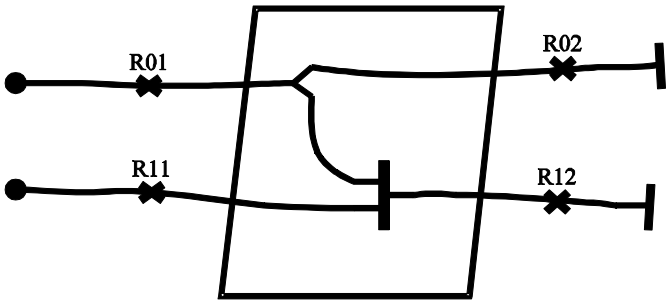


Figure 5.8: Example of implicit activities

UCM components

Components in a UCM describe the structure of a system. The class schema of Object-Z is a clear candidate to fulfill this role. It is, therefore, suggested to create a meta-class for any component that is not a team, as well as a hierarchy of meta-classes for each team component with one super-class, as well as and sub-classes.

Start point

A start point in a UCM is a place where triggering events occur to enable the execution of a scenario. Such events may possibly provide the system with some information that might need to be kept for further processing. In this regard, it is suggested adding, to the Z description of the UCM, an operation (e.g. $start(?)$), to consider the effect of such events on the system state.

End point

An end point in a UCM is a place where the execution of a scenario ends, enabling the resulting events to occur, and where post-conditions are gathered. They may therefore constitute, for example, in an operational system appropriate points to perform tasks such as: free temporary files, update log files, perform garbage collection and undo unachieved transactions. These constitute for plug-ins, the points from where the execution is returned to the main map.

5.4 Transformation process

This process assumes the use of one of the existing UCM traversal techniques, (Kealey and Amyot [47]), to scan an input UCM to identify individual map elements. This work considers only UCM elements discussed previously in Section 5.3 i.e. elements used in the case study; other UCM elements are beyond the scope of this dissertation. The core of the process

follows a bottom-up strategy that starts with the Z description of scenario paths, from their path segments, and the transformation of team components from their sub-components. If a map has no component, one implicit component for the system is assumed. In case the input map is complex or difficult to understand, before applying this process, it is recommended to use stubbing mechanisms to sub-divide the map into smaller and more manageable sub-maps.

The proposed transformation steps are:

Step 1- Construct Basic types, Abstract states, Operation schemas and Meta-classes: Initialise a list of basic types that will be populated progressively during the transformation of the input UCM.

1-1 For each UCM component that is not a team, specify a state schema to describe the part of the system state, controlled or represented by the component. When defining the invariant, consider relevant information such as the component's type, inter-component interactions, bounded scenarios, etc.

1-2 For each team component, recursively specify state schemas as follows: Create schemas for the contained components, and one schema for the container component. Combine these schemas using Z's schema calculus (e.g. schema inclusion or schema typing). Combining schemas aims to capture inheritance in a UCM. Where appropriate, use natural language prose to aid the specification.

1-3 For each stub, specify a state schema to describe the stub. Include in the predicate part as much information as possible that may help to relate the stub's incoming path segments, and the start and end points of plug-in(s). Add descriptive text wherever necessary for clarity.

1-4 For any other path element for which variables are required to keep information needed to describe their static or dynamic behaviour, create an appropriate Z state schema to model the state of such elements. For example, with a Timer, such a Z state schema may include a variable for the waiting time limit.

1-5 Complete the system state schema(s) and define realisable initial state(s) for the system.

1-6 For each path segment, create operation schemas to specify responsibilities (and other active path elements). In general, schemas for bound responsibilities, will apply to the local state of the binding component, but in some cases, they may apply to a larger, or even the whole system state.

1-7 For each start point, when necessary, create an operation schema, to consider the effect of the triggering events on the system state space.

1-8 For each end-point, when necessary, create operations to take into consideration resulting events and post-conditions, for the terminated scenario, in order to bring the system to a reusable state. With plug-ins, such operations should, for instance, return the execution to the main map. The list of those operations, for a given plug-in, would therefore constitute the component *returnPoints* (discussed earlier).

1-9 Create schemas for control operations, (see Section 5.3 above on stubbing techniques and active components), associated to each active component.

1-10 Use schema composition to construct a sequence of schemas, that will describe scenarios over a full path, (a sequence of path elements). Also consider path elements and path connectors.

1-11 When necessary, create, a meta-class for each component (an Object-Z class for which properties and methods are not yet defined), that will in later stages encapsulate both the state and operations performed by the component.

Step 2- Complete the Z schemas, and generate Object-Z class schemas (Periyasamy and Mathew [68] provide more details on mechanisms to transform Z schemas into Object-Z).

2-1 Calculate preconditions for important operations to generate partial operations for error conditions. At this point, the calculated preconditions may help to improve the input UCM. One also may employ guiding principles for constructing Z schemas (van der Poll et al. [94]), where appropriate.

2-2 Define total operations (covering error conditions) corresponding to each partial operation, defined in Step 21 above.

2-3 Complete each meta-class with appropriately selected schemas. In general, those schemas must have been generated from elements of path segments, that are bound to the component.

5.5 Chapter summary

This chapter has demonstrated the transformability of UCMs, by proposing a generic framework to translate a UCM, into a Z, and an Object-Z, specification. The suggestion consisted to initially transform the functional or behavioural aspects of the input map, into Z schemas, and its architectural components into meta-classes of Object-Z. Afterwards, combine the Z schemas and meta-classes to obtain Object-Z classes. The fundamental Object-Oriented concepts of inheritance and encapsulation are used to capture information on the structuring of

UCM components. Concepts in UCM, Z and Object-Z (and possible relationships between them), were analysed leading to a conclusion that a one-to-one transformation may be hard to achieve, since the notations involved, operate at different levels of detail. Nevertheless, the proposed description of UCM concepts in Z, and Object-Z, may constitute a reasonable move in Requirements Engineering, as this may provide designers with key important benefits through the use of formal methods in systems engineering. In fact, the main ideas proposed here, were summarised in a full research paper, presented at the 7th International Workshop on Modelling, Simulation, Verification and Validation of Enterprise Information System in 2009 (MSVVEIS 2009). Full details may be reached in (Dongmo and van der Poll [23], P. 3-13).

The next chapter, demonstrates the applicability of this framework, by applying it to the UCMs of the Case Study developed in Chapter 3, and aims to produce the UCM-OZ version, of the Object-Z specification, of the case study.

Chapter 6

Applying the UCM transformation framework

This chapter applies the UCM transformation framework proposed in Chapter 5, to the UCM model of the case study developed in Chapter 3, to generate an Object-Z specification. The input map is presented in Section 6.1, and the stubbed version of the same map is developed in Section 6.2. In Section 6.3, the stubbed Map is transformed to Z and Object-Z. The chapter concludes with a brief summary in Section 6.4.

NB: Due to the automatic generation of references by Latex, references to guidelines from Chapter 5 are presented as follows: e.g. the original *Guideline*#1 − *xy* becomes *Guideline*#1*xy* without the (−) separator.

6.1 The input UCM map

This section presents the input map from Chapter 3 that will be transformed, in this chapter, to an Object-Z specification using the guidelines proposed in Chapter 5. The descriptive text used to explain some aspects of the map, is not reproduced, as it is accessible from Chapter 3. To facilitate the comprehension and the transformation process of the input UCM, presented in Figure 6.1, a stubbing mechanism is used to sub-divide it into sub-maps (as recommended in the framework presented in Chapter 5, Section 5.4). Four different maps, therefore, result from sub-dividing the input UCM: the main map presented in Figure 6.2, including a static stub, namely, *NetControl*, and a dynamic stub, namely, *Validate*. The other three maps are: the plug-in in Figure 6.3 associated to the static stub, and two alternative plug-ins (see respectively, figures 6.4 and 6.5 associated to the dynamic stub). The following names are proposed for the abstract components that were unnamed in the original map: *Pay_point*, *Transit_point*, *Check_point*, and *Update_point* (Figure 6.2). Those names are for referencing purposes.

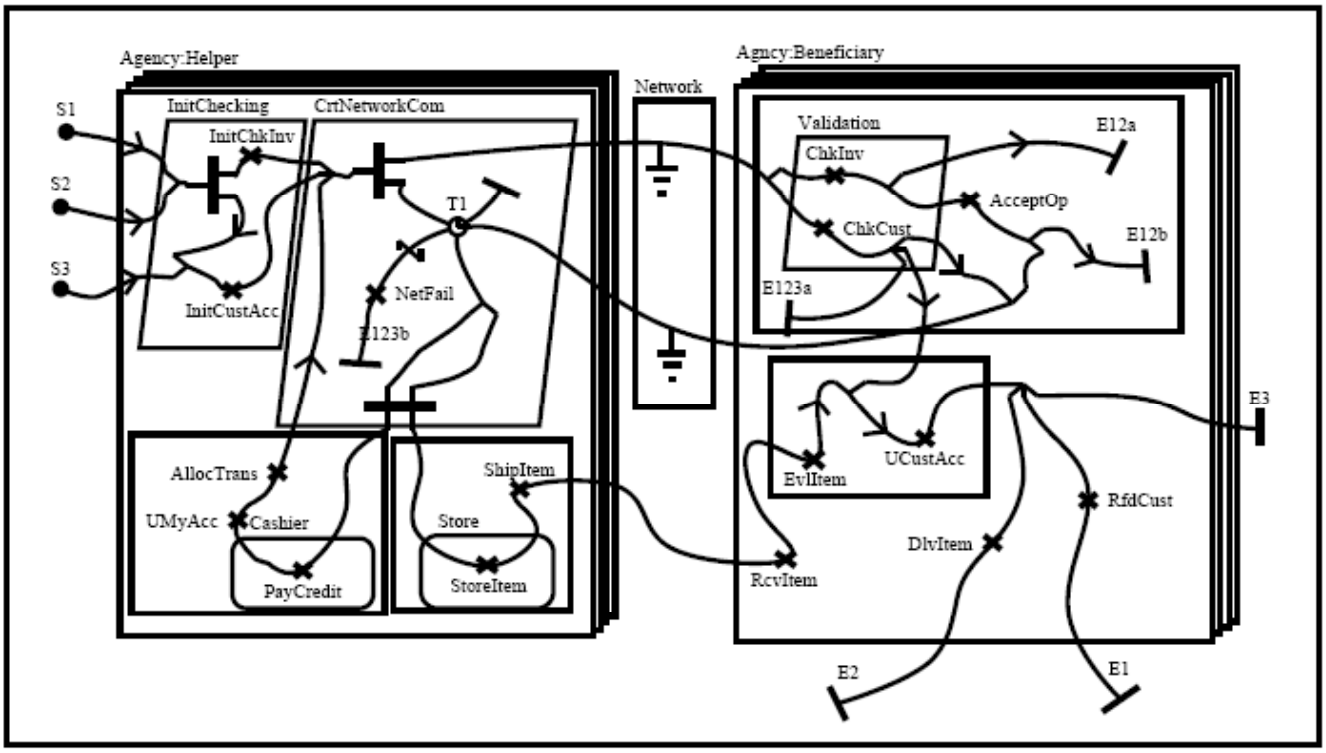


Figure 6.1: Initial UCM map

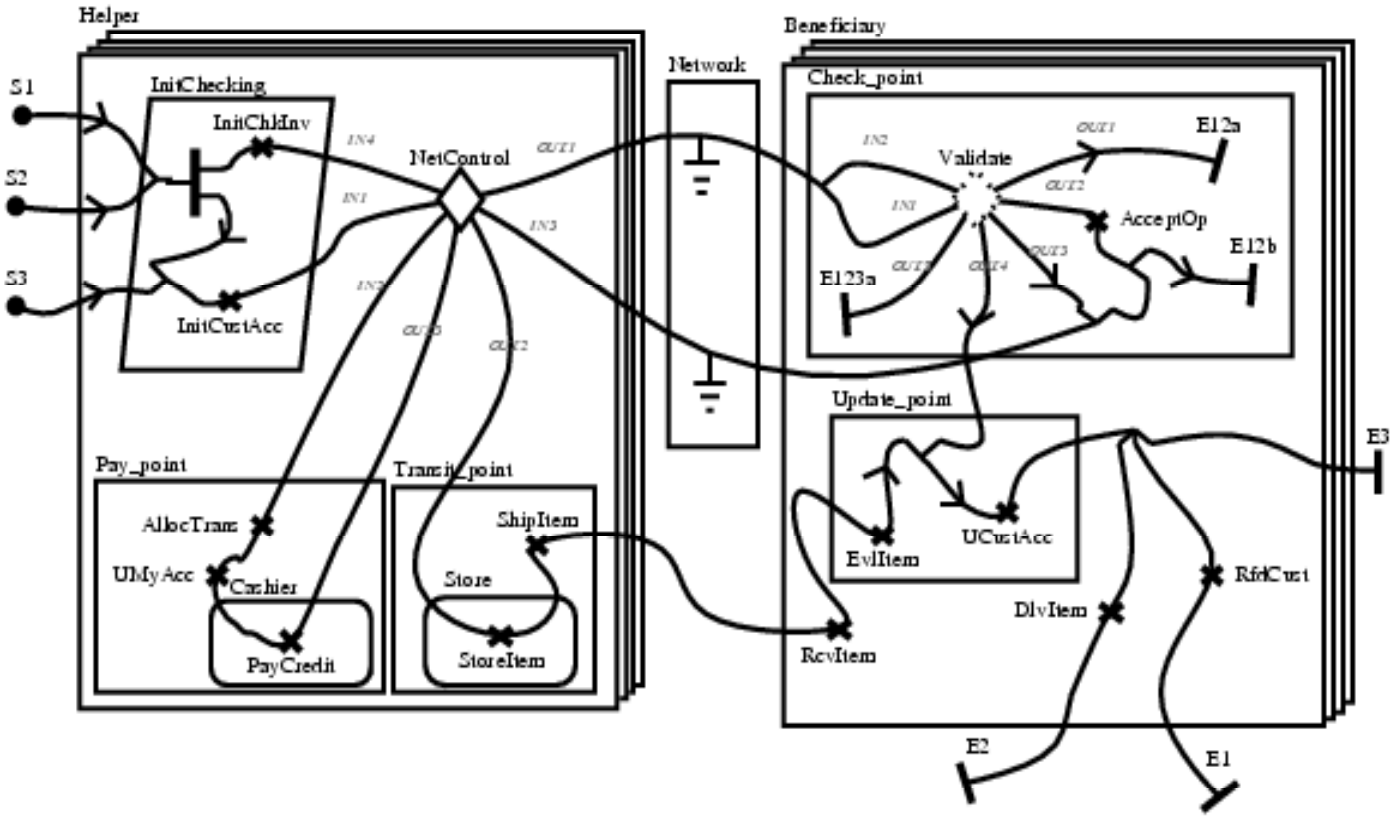


Figure 6.2: Stubbed UCM map

6.2 The stubbed UCMs

In the main map presented in Figure 6.2, the static stub named **NetControl** represents a place where the sub-map (called a plug-in) shown in Figure 6.3, fits during runtime, to perform the task of forwarding incoming requests to a beneficiary agency through the network component. It includes four input segments, namely $IN1$, $IN2$, $IN3$ and $IN4$ through which stimuli flow from the main map to the plug-in, and three output segments named $OUT1$, $OUT2$, and $OUT3$, through which the plug-in forwards the result of its execution to the main map.

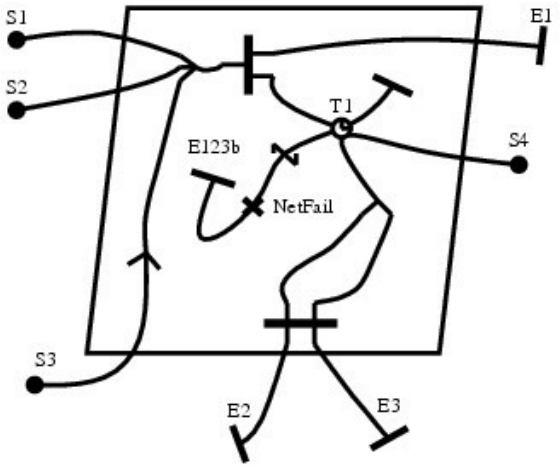


Figure 6.3: Expansion of the NetControl stub in Figure 6.2

Similar to the stub, the plug-in (see Figure 6.3), associated with the static stub, has four start points connected to the four input segments of the stub. It also has three end-points which are connected to the output segments of the stub. The end-point labelled E123b (Figure 6.3) may be reached during runtime to terminate a scenario, after the responsibility point labelled "NetFail", which is performed responding to a timeout event. A timeout event occurs when the maximum waiting time elapses, before any response for a remote request arrives. Within the main map, the binding relationship that statically links the input path segments to start-points of the plug-ins, and the end-points of the plug-in to the output path segments of the stub, are denoted in UCM as follows:

- $\langle IN1, S1\rangle, \langle IN2, S2\rangle, \langle IN3, S3\rangle, \langle IN4, S4\rangle$. Each such binding, for example between $E1$ and $S1$, indicates that the start point $S1$, is triggered by a stimulus from the input segment $E1$.

- $\langle E1, OUT1\rangle, \langle E2, OUT3\rangle, \langle E3, OUT2\rangle$. The binding relationship between $E1$ and $OUT1$, for example, indicates that a resulting event at the end-point $E1$ flows to the main map, via the output segment $OUT1$.

The plug-in in Figure 6.3 implements a timeout recovery mechanism to control the transmission of requests over a network. Any request that arrives at the plug-in by triggering one of the start points $S1$, $S2$, or $S3$, is re-transmitted to an appropriate agency via the binding channel $\langle E1, OUT1\rangle$, after the timer is setup with an appropriate time limit. If a response for a particular request is not received, through $\langle IN4, S4\rangle$, before timeout occurs, the process within the plug-in performs the responsibility named "NetFail", and terminates the scenario. Otherwise, the sub-map transmits the resulting event at $E2$ or $E3$, depending on the request, to the main map, via the links $\langle E2, OUT3\rangle$ or $\langle E3, OUT2\rangle$. The sub-map was indeed extracted from the initial input UCM (see Figure 6.1), and the semantics of the path elements remain unchanged (as described in Chapter 3, Section 3.3.3).

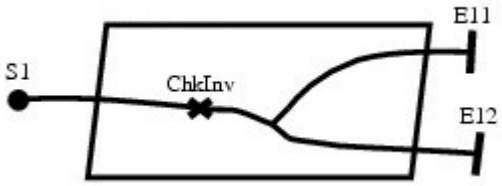


Figure 6.4: Validate an invoice

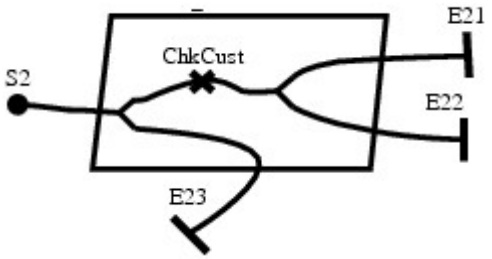


Figure 6.5: Validate a customer

The dynamic stub in the main map (Figure 6.2), may select either the plug-in (in Figure 6.4) to check an invoice, or that of Figure 6.5, to validate a customer. The stub includes two input and five output segments. The selection of a plug-in during runtime depends on the input segment from which a stimulus reaches the stub. The dynamic binding relationship that links the input segments of the main map to start points on the sub-maps and those that link the end points of the sub-maps (figures 6.4 and 6.5), to output segments of the main map (Figure 6.2), are presented in UCM as follows:

- $\langle IN1, S1\rangle, \langle E11, OUT1\rangle, \langle E12, OUT2\rangle$. The sub-map in Figure 6.4 is selected when a stimulus comes from $IN1$.

- $\langle IN2, S2\rangle, \langle E21, OUT5\rangle, \langle E22, OUT3\rangle, \langle E23, OUT4\rangle$. The plug-in in Figure 6.5, is selected when a stimuli comes from the path segment, labelled $IN2$.

The sub-maps (plug-ins)(figures 6.3, 6.4 and 6.5) and the main map (Figure 6.2) are extracted from the initial UCM map (Figure 6.1). Therefore, the prose text descriptions given in Chapter 3, Section 3.3.3, to explain UCM elements, remain valid for each element of the main map and plug-ins.

The binding relationship between the stub, in the main map, and the two alternative plug-ins, indicates that: a request to check an invoice reaches the stub via $IN1$, and is performed by the plug-in of Figure 6.4. A request to validate a customer, or to update a customer's account, arrives to the stub, via $IN2$, and is performed by the plug-in in Figure 6.5. When the validation of an invoice fails, the system follows to $E11$; otherwise, it continues to $E12$ (Figure 6.4).

The plug-in forwards all requests to update a customer's account to the end point $E23$, and when a request is received to validate a customer, the plug-in performs the responsibility point labelled **ChkCust**(see Figure 6.5), to validate a customer. If such a validation fails, the system continues to the end point $E21$, otherwise, it follows the path segment to the end point $E22$.

In the next section, guidelines proposed in the framework of Chapter 5 are followed to transform the stubbed UCM, so far described, into Z and Object-Z (see Figure 1.3).

6.3 The Z and Object-Z specifications

As indicated in Figure 6.6, the original UCM has been decomposed into four sub-maps including one main map, namely, the Stubbed UCM, that represents the UCM in Figure 6.2, and three plug-ins.

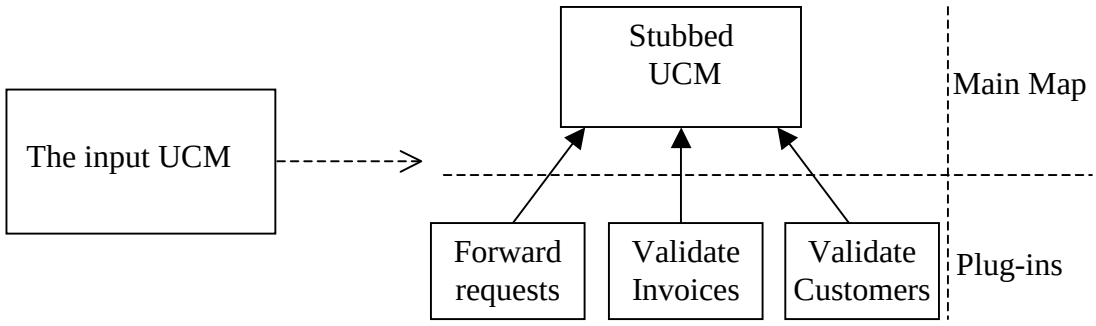


Figure 6.6: Decomposing the input UCM

There is also a plug-in to forward requests over the network (shown in Figure 6.3), namely, **NetControl**, and two alternative plug-ins for the dynamic stub: one to Validate Invoices (shown in Figure 6.4) and the other to Validate Customers (shown in Figure 6.5). The arrows from plug-ins to the main map indicate the binding relationship between each plug-in and the stubbed map. To accommodate the "bottom-up" strategy adopted in the transformation framework (see Chapter 5), the sub-maps, which are included in the main map, are first transformed, followed by the main map.

Despite the fact that sub-maps may be treated separately, as suggested in Chapter 5, Section 5.4 on page 89, they remain part of the main map; the principal reason for the stubbing mechanism is to bring clarity into large or complex maps. For this reason, for the Z transformation, only one list of basic types, and one of the global variables for the system, (including the main map and sub maps), are presented in Section 6.3.1. Each of the lists is obtained by applying the instruction # 11 of the framework (see Section 5.4, Chapter 5), to each of the UCM maps. In Section 6.3.1, those variables are encapsulated into a single class schema named *ClsGlobalVariables*, making it possible for other classes to inherit it, when needed. An overview of the transformation process is presented below:

- Section 6.3.1: Generate basic types, and global variables;
- Section 6.3.2: Apply the transformation framework to the plug-in, to forward requests;
- Section 6.3.3: Apply the transformation framework to the plug-in, to validate invoices;
- Section 6.3.4: Apply the transformation framework to the plug-in, to validate customers;
- Section 6.3.5: Apply the transformation framework to the stubbed map taking into consideration the binding relationships with the plug-ins.

Next, the given sets and variables for the entire system are presented.

6.3.1 Given sets and global variables

A set of given types for the whole system (including the main map and the three plug-ins) is given below:

$$[Request, Identifier, Customer, Account, Invoice, Address, Item, Agency, \\ Money, Date, Message, Time]$$

Request represents the set of all possible requests. *Identifier* is the set of all possible identifiers. The types *Customer*, *Account*, and *Invoice* represent, respectively, the set of all

possible customers, accounts, and invoices. Similarly, *Address*, *Item*, *Agency*, and *Money* are respectively, the sets of all possible addresses, items, agencies that may be involved, and all possible amounts of money, with associated currencies. The set *Date* is used for all possible dates; *Message* is the set of all possible messages that may be exchanged between the system and the environment, and *Time* is the type of possible times.

Next the definition of the datatypes is presented.

$$Response ::= Ok \mid Failed \mid Accepted \mid Denied \mid Unknown$$

The balance of an account is defined as a partial function, that associates the account to an amount of money.

$$balance : Account \pfun Money$$

The function *debit* is needed, for example, to issue a payment into an account.

$$debit : Account \times Money \pfun Account$$

$$\begin{aligned} &\forall\, acc, acc' : Account;\ amnt, m : Money \bullet \\ &\quad ((acc, amnt) \mapsto acc' \in debit) \Leftrightarrow \\ &\quad ((acc, m) \in balance \Rightarrow (acc', m + amnt) \in balance) \end{aligned}$$

The partial function *addressOf*, maps each agency to its address.

$$addressOf : Agency \pfun Address$$

It is assumed each item has a value at a particular given time. The function *itemsValue* maps each item to its present value.

$$itemsValue : Item \pfun Money$$

The partial function *reqOption* uniquely maps to each request, which involves a purchased item, the invoice that includes the item.

$$reqOption : Request \pinj Invoice$$

There are three types of scenarios as defined below:

$$Scenario == \{sceneReturnItem, sceneReplaceItem, scenePayCredit\}$$

Since a request is issued only when a particular scenario is in progress, the same request may not be shared by more than one scenario. So, the identifier associated with a request may be used to identify the scenario. In this regard, the function *idScenario* is defined.

$$idScenario : Identifier \pfun Scenario$$

The class schemas representing the Object-Z transformation of the basic types and variables thus far defined, are presented next.

Class schemas for basic types and variables

The Object-Z transformation of a basic type, yields a class with no state and no operation (see Dongmo and van der Poll [23], Periyasamy and Mathew [68]). The reason is that a given type, describes objects of which properties and methods are undefined. So, each of the types defined above, is transformed in Object-Z, into such class schema. Next, an example is shown of a graphical representation of the class corresponding to the transformation of the set *Request* into Object-Z. Other such classes may be similarly represented.

Request

For these classes, we prefer to keep the same name in Z and Object-Z. The purpose is to maintain the coherence between the Z and the Object-Z versions.

The class schema, represented below, is created to encapsulate the global variables described above, to make them accessible to other classes, for example, by aggregation or inheritance (see Duke and Rose [26]).

ClsGlobalVariables

$$\lceil(addressOf, itemsValue, balance, debit, Response, Scenario, invoiceInReq, customerInReq)$$

$$addressOf : Agency \pfun Address$$

$$itemsValue : Item \pfun Money$$

$$invoiceInReq : Request \pfun Invoice$$
$$customerInReq : Request \pfun Customer$$

$$\mathrm{dom}\ invoiceInRequest \cap \mathrm{dom}\ customerInReq = \varnothing$$

$$idScenario : Identifier \pfun Scenario$$

$$balance : Account \pfun Money$$

$$Response == Ok, Failed, Accepted, Denied, Unknown$$

$$debit : Account \times Money \pfun Account$$

$$\forall acc : Account; acc' : Account;\ amnt, m : Money \bullet$$
$$((acc, amnt) \mapsto acc' \in debit) \Leftrightarrow$$
$$((acc, m) \in balance \Rightarrow (acc', m + amnt) \in balance)$$

$$Scenario == \{sceneReturnItem, sceneReplaceItem, scenePayCredit\}$$

Next, a transformation into Z and then, Object-Z of the sub-map is proposed (see Figure

6.3), which is associated with the static stub, on the main map in Figure 6.2.

6.3.2 Applying the framework to the plug-in to forward requests

This plug-in is activated when one of the start points $S1$, $S2$, or $S3$ is triggered. An incoming segment is triggered when the progression of a scenario along a path reaches the sub-map. In the plug-in under consideration, the purpose of such a triggering event may be to request the validation of an invoice, the validation of a customer, or to request the update of a customer's account. Due to the fact that the given sets and global variables for the whole system are readily available (see Section 6.3.1), and may be used wherever necessary in the system, they do not need to be redefined. The next step, is to describe the possible Z states that may result from transforming the plug-in, to a Z specification. In this regard, the UCM elements included in the sub-map are considered in the next paragraph.

The UCM elements encountered when traversing the map from start to end points are respectively:

- Four start points: $S1$, $S2$, $S3$ and $S4$,
- A process (representing a UCM abstract component),
- An OR-join (joining together the path segments from $S1$, $S2$ and $S3$),
- An AND-fork (that synchronises the progression of a scenario along the path segment to $E1$ and the timing operations),
- A Timer $T1$,
- An OR-fork (it splits the path segment from $S4$ into two: one through which is forwarded a response for a request to validate an invoice, and another path through which a request to validate a customer is forwarded),
- An AND-join (to ensure that for some scenario, both the invoice and customer are successfully validated before progressing towards the end point $E3$), and
- Three end points: $E1$, $E2$ and $E3$.

In the light of the analysis, relative to the above listed UCM elements, performed in Chapter 5, Section 5.3 and the transformation guidelines # 11 to 15 which depict the Z states of a UCM, the abstract state spaces resulting from this sub-map are suggested.

Abstract state definition

The state schema *netComTemp* is created for the UCM abstract component *Process*, to temporarily keep information on pending requests until they are processed. For that reason, it will be referred to as a "temp" file. The schema derives from *Guideline # 11* (Chapter 5, Section 5.4), that suggests creating a state space for each abstract component other than a team component.

netComTemp

$$
\begin{array}{l}
reqInvoices : Identifier \rightarrowtail Invoice \times Address \\
reqCustomers : Identifier \rightarrowtail Customer \times Address \\
reqTransactions : Identifier \rightarrowtail Customer \times Money \times Address \\
reqRespones : Identifier \nrightarrow Message \\
\hline
\mathrm{dom}\ reqInvoices \cap \mathrm{dom}\ reqCustomers = \varnothing \\
\mathrm{dom}\ reqInvoices \cap \mathrm{dom}\ reqTransactions = \varnothing \\
\mathrm{dom}\ reqCustomers \cap \mathrm{dom}\ reqTransactions = \varnothing \\
\mathrm{dom}\ reqResponses \subseteq \mathrm{dom}\ reqInvoices \cup \mathrm{dom}\ reqCustomers \cup \mathrm{dom}\ reqTransactions
\end{array}
$$

The variable *reqInvoices* keeps a list of mappings in which, for each record, an identifier is uniquely mapped to a product of an *Invoice*, with an *Address*. The invoice requires to be validated, and the address is that of the agency from which the invoice was issued. The variable *reqCustomers* records a list of mappings between identifiers and products of a customer and an address. The customer, who holds the invoice, also needs to be validated and the address is that of the agency where the customer's account is. The information maintained in those two variables is used to support the activities of the system when returning or replacing items. The variable *reqTransactions* is helpful to the system when a customer is paying a credit. Each record in it contains an identifier that is used during the entire process to uniquely identify the product of the three objects needed: the customer who is paying (*Customer*), the amount of money involved (*Money*), and the address of the target agency that holds the customer's account that needs to be updated. The partial function *reqResponses* associates each response, to an identifier that may be used to trace a record in one of the above mentioned components. The predicate part indicates that an identifier is not allowed to reference more than one request, and a response may be provided only for a pending request.

The next state schema *Timer*, is created to describe the state of the timer (*Guideline # 14*).

Timer

$$
\begin{array}{l}
opened : Boolean \\
waitingId : Identifier;\ maxtime : Time \\
\hline
opened = true \Rightarrow waitingid = \bot \wedge maxtime = 0
\end{array}
$$

The variable *opened* indicates the status of the *Timer*. For example, when the timer is not activated, the value of *opened* is true, or conversely false otherwise. When the timer is not activated, the variable *waitingId* that contains a request identifier, is undefined. This schema results from the analysis performed in Section 5.3 of Chapter 5.

In the next section, the effect of the AND-fork connector on the system, is presented as a general theory (Potter et al. [70]).

Some general theory: AND-fork

An AND-fork path element in a map synchronises the timing activities and the execution of the path segment, between the element and the end-point $E1$ (Figure 6.3). Although no responsibility point is placed on this path segment, an implicit activity is to be considered, for the reason that, at $E1$, a request must have been sent over the network. In this regard, three operations are created in Section 6.3.2, to handle respectively, the sending of requests over the network to:

a) check an invoice (*reqCheckInvoice*),

b) validate a customer (*reqCheckCustomer*), and

c) update a customer's account (*reqUpdateAccount*).

Since the plug-in (Figure 6.3) sends all the requests via the network component (Figure 6.2), a state schema is required to specify the communication interface between the plug-in, and the network. The interface illustrates the coupling between the plug-in (and also the static stub in Figure 6.2), and the network component. This coupling is revealed by the path segment, (incoming path segment $IN4$) that joins the static stub (Figure 6.2) with the network component. The state schema presented next, is created in line with the *Guideline # 11.*

interface

$$reqToForward : Identifier \rightarrowtail\!\!\!\!\!\rightarrow Request$$
$$reqToReceive : Identifier \rightarrowtail\!\!\!\!\!\rightarrow Request$$

The state space *interface*, groups the two main components of the network made available to the environment and *netInterface*; the schema presented next, is more generic, as it includes constraints on components, hence, linking interfaces to each other.

netInterface

interface

$$(\forall\, id \mapsto req \in reqToReceive)(\exists_1\, Interface \mid id \in \operatorname{dom} \theta Interface.reqToFoward)$$

The variable *reqToFoward* contains the list of requests sent via the network, and *reqToReceive*, the set of requests waiting to be retrieved at a network terminal. The predicate indicates that a request received at a network was previously sent through another network interface.

At this stage, the state of the system as a whole includes the three state schemas defined above.

requests

$$\begin{array}{|l} netComTemp \\ timer \\ netInterface \\ \hline waitingId \in \operatorname{dom} reqInvoices \cup \operatorname{dom} reqCustomers \ \cup \operatorname{dom} reqTransactions \\ \hline \end{array}$$

As recommended by the enhanced strategy for documenting a Z specification (Lightfoot [52], Potter et al. [70]), the next section presents some initial states for the above defined, abstract states.

Initialisation

It is assumed that the system starts with both *netComTemp* and *netInterface* empty, and the timer open.

InitNetComTemp

$$\begin{array}{|l} netComTemp' \\ \hline reqInvoices' = \varnothing \land reqCustomers' = \varnothing \land reqTransactions' = \varnothing \\ \hline \end{array}$$

The condition $reqResponses' = \varnothing$ may be deduced from the values of the three components included in the initial state. For brevity (Gravell [34]), it is therefore taken out of the schema.

InitTimer

$$\begin{array}{|l} timer' \\ \hline opened' = true \land waitingid' = \bot \land maxtime' = 0 \\ \hline \end{array}$$

Note, that having a component equal to an undefined values (e.g. $waitingid' = \bot$), simply means that the value of the component is not yet defined, or is unknown. Although the notation may be misleading if minterpreted, it may equally bring more clarity (Gravell [34]) into a specification, and help to avoid some common mistakes that occur in error conditions (van der Poll and Kotzé [93]).

InitNetInterface

$$\begin{array}{|l} netInterface' \\ \hline reqToFoward' = \varnothing \land reqToReceive' = \varnothing \\ \hline \end{array}$$

Next the partial operations for the part of the system modelled by the UCM in Figure 6.3 are discussed.

Partial operations

This section starts with the description of operations associated with the start points (see *Guideline # 17*, Chapter 5). By convention, the name of each of those operations is the same as that of the start point (but in lower case, as the names of start points are in upper case).

The start points $S1$, $S2$ and $S3$, are respectively triggered to request the validation of an invoice, a customer account, or the updating of a customer's account. The associated operations $s1$, $s2$ and $s3$ capture the input information from the environment and keep them temporarily during the scenario execution. They also ensure that a unique identifier is attached to each pending request.

$s1$

$$\Delta netComTemp$$
$$id? : Identifier;\ inv? : Invoice;\ ad? : Address$$

$$id? \notin \text{dom}\, reqInvoices \cup \text{dom}\, reqCustomers \cup \text{dom}\, reqTransactions$$
$$reqInvoices' = reqInvoices \cup \{id? \mapsto (inv?, ad?)\}$$

$s2$

$$\Delta netComTemp$$
$$id? : Identifier;\ customer? : Customer;\ ad? : Address$$

$$id? \notin \text{dom}\, reqInvoices \cup \text{dom}\, reqCustomers \cup \text{dom}\, reqTransactions$$
$$reqCustomers' = reqCustomers \cup \{id? \mapsto (customer?, ad?)\}$$

$s3$

$$\Delta netComTemp$$
$$id? : Identifier;\ customer? : Customer;\ amnt? : Money$$
$$ad? : Address$$

$$id? \notin \text{dom}\, reqInvoices \cup \text{dom}\, reqCustomers \cup \text{dom}\, reqTransactions$$
$$reqTransactions' = reqTransactions \cup \{id? \mapsto (customer?, amnt?, ad?)\}$$

The condition given by the expression

$$id? \notin \text{dom}\, reqInvoices \ \cup\ \text{dom}\, reqCustomers \ \cup\ \text{dom}\, reqTransactions$$

in the predicate part of each of the three operations defined so far, ensures that the input identifier $id?$ is not currently associated with any pending request. Due to the fact that the

expression is repeated in each operation, it is arguable that, for brevity, it may be given a name and be referenced in the operation, by that name. Nevertheless, it is preferred to repeat the expression, so as to make its meaning clear (Gravell [34, Section 2.1]) while reading the specification.

The start point $S4$, is triggered when a message from the network, referenced by a request identifier, reaches the plug-in. The operation $s4$ is activated to commit the message into the temp file. The operation is described as:

$s4$

$\Delta netComTemp$
$id? : Identifier;\ resp? : Message$

$id? \in \text{dom}\ reqInvoices \cup \text{dom}\ reqCustomers \cup \text{dom}\ reqTransactions$
$reqResponses' = reqResponses \cup \{id? \mapsto resp?\}$

The input variable $id?$ contains the identifier of the request for which the incoming response in $resp?$ is passed as input to the operation. The system uses the two inputs to update the list $reqResponses$.

Next, the schema of an operation *send* provided by the network interface, to send requests over the network, is considered.

send

$\Delta netInterface$
$id? : Identifier;\ req? : Request$

$reqToFoward' = reqToFoward \cup \{id? \mapsto req?\}$

This operation is provided by the network interface to facilitate the sending of messages through a network. Although the operation is not explicitly represented as a responsibility point on the initial UCM, it results from the natural activities of the component *Network* (Figure 6.2), which is to serve as an active support for a bi-directional communication between agencies (see Figure 3.1). The operation is therefore created in line with *Guideline #16* in the transformation framework (Chapter 5, Section 5.4) to update the list *reqToFoward* of requests. The list contains those requests that are being transmitted by the network, from one agency to another.

The three operations of the plug-in that use the interface operation *send* to forward the three types of requests over the network, are defined next. Each generates a request for which an identifier is provided as an input. The identifier and the request are passed to the

interface via the operation *send*, that handles the responsibility to transmit the request to its destination via the network.

As mentioned above, the operation *reqCheckInvoice* presented next, submits a request to check an invoice.

reqCheckInvoice

$$\Xi netComTemp$$
$$\Delta netInterface$$
$$id? : Identifier$$

$$id? \in \operatorname{dom} reqInvoices$$
$$\exists\, req : Request \bullet Send(id?, req)$$

This operation is applicable when the input *id*? is a valid identifier associated to an invoice in *reqInvoices*. The system generates a request, and sends it to the appropriate agency by the means of the interface operation *Send*. Note, that the use of the operation *Send* in the expression of the predicate, may be suspected because an operation does not explicitly represent a logic expression. Such constructions are, however, allowed, since similar cases are found in the literature, notably in Bowen [13] (when defining a schema operation for sequential composition of two operations) and Potter et al. [70] (when describing the Schema hiding operators, e.g. $EnterNewCopy \mathrel{\widehat=} \exists\, c? : Copy \mid c? \notin \operatorname{dom} stock \bullet ToStock$ where *ToStock* is an operation).

Similar to *reqCheckInvoice*, the operations *reqCheckCustomer* and *reqUpdateAccount*, respectively, generate requests to check a customer, and update customer accounts, and send them through the network interface using the operation *send*.

reqCheckCustomer

$$\Xi netComTemp$$
$$\Delta netInterface$$
$$id? : Identifier$$

$$id? \in \operatorname{dom} reqCustomers$$
$$\exists\, req : Request \bullet send(id?, req)$$

reqUpdateAccount

$$\Xi netComTemp$$
$$\Delta netInterface$$
$$id? : Identifier$$

$$id? \in \operatorname{dom} reqTransactions$$
$$\exists\, req : Request \bullet send(id?, req)$$

At this stage, when necessary, one may consider to improve the original UCM by adding to it, the three operations described previously (as the double-headed arrows in Figure 5.1, Chapter 5 suggests). For example, they may be added as responsibility points before or after calculating their preconditions.

The general form of the schema calculus expression that defines the operations related to a timer, was given (in Chapter 5, on page 82). The formula is:

$$(schemaRin \land blockTimer) \fatsemi ((schemaCin \lor schemaTout) \land release) \fatsemi schemaRout$$

The schema *schemaRin* describes operations performed by the system, immediately before sending a request and activating the timer. In the case of the present UCM, this operation may be omitted because no such operation is performed by the plug-in.

blockTimer was given as: $blockTimer = block \land setup$ with *block* and *setup* defined as:

block

$$\Delta Timer$$
$$id? : Identifier$$
$$opened = true \land waitingId' = id? \land opened' = false$$

This operation is allowed when the timer is opened. The identifier associated to the request is passed to the variable *waitingid*, and the timer is blocked ($opened' = \text{false}$).

setup

$$\Delta Timer$$
$$time? : \mathbb{N}$$
$$time? > 0 \land maxtime' = time?$$

This schema specifies an operation to attribute a value to the maximum waiting time variable, *maxtime*.

The schema calculus expression *schemaCin*, is constructed from the operations that describe the reaction of the system when an event occurs on the clearing path (see Chapter 5). Such an event reaches the plug-in, via the binding relationship $\langle In4, S4 \rangle$ (see Section 6.2, Page 95). In this particular case, the operation $s4$, associated with the start point, is performed to temporarily maintain the input data provided by the environment. The operation *respond* described below, is performed to allow the timer to be unblocked.

respond

$\Xi timer$

$id? : Identifier;\ resp?, resp! : Message$

$id? = waitingId \land resp! = resp?$

When the value of *id?* is the same as the one held in *waitingId*, the incoming message from the network is made available to the user, and the timer is unblocked. Otherwise, the system keeps on waiting.

The schema *schemaTout* (see equation 6.3.2) is intended to describe the reaction of the system when a timeout event occurs. In the case of this system, the operation *netFail* is performed to recover from the failure network. In practice this operation may become very complex depending, for example, on the criticality of the problem under consideration. For instance, a late response in a medical application system would not have the same impact as in an ordinary emailing system. In this work, the *netFail* specifies a warning message for the user.

netFail

$\Xi Timer$

$Resp! : Message$

$maxtime \leq 0 \land resp! = Timeout$

The operation *release*, unblocks the timer and makes it available. It follows a successful execution of *respond* or *netfail*.

release

$\Delta Timer$

$opened' = true \land waitingId' = \bot \land maxtime' = 0$

The schema *schemaRout*, from Equation 6.3.2, is meant to describe any operation performed immediately after the timer is released. It may be omitted because, in the UCM in Figure 6.3, no explicit activity takes place on the main path, between the timer and the next path element.

The schema expression for the timing operations becomes:

$$opTimer = blockTimer \fatsemi (((s4 \fatsemi respond) \lor netFail) \land release) \tag{6.1}$$

The sending of requests may also be summarised as:

$$sendRequest = reqCheckInvoice \lor reqCheckCustomer \lor reqUpdateAccount \tag{6.2}$$

where the schemas of the operations *reqCheckInvoice*, *reqCheckCustomer* and *reqUpdateAccount* were defined above. The next three sections illustrate, respectively, the influence of the path connectors: AND-fork, OR-join and AND-join in the Z description of the operations of the system as modelled with the input UCM, in Figure 6.3.

The AND-fork

Because of the AND-fork connector (placed before the timer), a scenario is allowed to continue only after the sending of a request, as well as the timing operations, have both been completed. That is, when the operation defined by the schema expression 6.3, is successfully performed.

$$sendRequest \wedge opTimer \tag{6.3}$$

The OR-join

This path connector allows scenarios to share the path segment placed after it. A generic operation schema, that includes the effect of an OR-join path element, in combination with the operations along the incoming and outgoing path segments, was given in Chapter 5 by the schema expression:

$$(Op11 \vee Op12) \fatsemi Op2 \tag{6.4}$$

Where $Op11$, $Op2$ and $Op2$ respectively describe operations on the first and second incoming path segments, and the operation on the outgoing path segment (see Figure 5.3). With the input UCM (Figure 6.3), there are three incoming path segments, and the only operations bound to them are those that were described to handle triggering events on start points $S1$, $S2$ and $S3$. Hence, in line with *Guideline # 110* of the framework in Chapter 5, in this transformation process toward the construction of the sequence of activities performed along paths, the expression 6.4 may therefore be instantiated to:

$$(s1 \vee s2 \vee s3) \fatsemi ((sendRequest \wedge opTimer) \fatsemi op) \tag{6.5}$$

The Schema expression *op*, describes the operations performed when the Timer is released. The composite activities *SendRequest* and *opTimer* were respectively defined with the formulas 6.1 and 6.2. Although the schema calculus expression in 6.5 illustrates the influence of the path connector on the system operations along paths, further consideration may be made when defining an individual operation, on the outgoing path segment. When necessary, in the predicate part of the control module, or in operation encountered on the outgoing path segment, a disjunct clause, or an if/then/else statement, should be included to enable the system to distinguish which scenario is in progress along a shared path.

The OR-fork

This path connector is placed immediately after the timer, and before the AND-fork (see Figure 6.3. It allows the execution of a scenario to follow one of the two outgoing paths depending on whether the current request is to validate an invoice, or a customer's account. As in the case of the OR-join above, the generic schema expression 6.6 below, and suggested in Chapter 5, illustrates the influence of the path elements on the structuring of the sequence of system operations along paths.

$$Op1 \fatsemi (Op11 \lor Op12) \tag{6.6}$$

The schema $Op1$ specifies all the operations in the incoming path segment, and the schemas $Op11$ and $Op12$ each specify the operations along an outgoing path segment. Referring to the input UCM in Figure 6.3, in line with *Guideline # 110* of the proposed framework in Chapter 5, this would therefore extend the schema expression in 6.5, to includes $Op11$ and $Op12$:

$$(s1 \lor s2 \lor s3) \fatsemi ((sendRequest \land opTimer) \fatsemi (Op11 \lor Op12)) \tag{6.7}$$

The two operations $Op11$ and $Op12$ are described in subsequent sections.

The AND-join

This path connector is the last element, before the end-points $E2$ and $E3$ (see Figure 6.3). It influences the system only when returning or replacing an item. It helps to ensure that both the invoice, and customer, have been successfully validated before the system can continue to the end point $E3$. When paying a credit, the scenario progresses directly to the end-point $E2$. Because this connector impacts two instances of the same plug-in (one that checks the invoice, and the other that checks the customer), its effect would be reflected in the predicate part of the control operation, for example in the form of a conjunct clause.

In line with *Guideline # 18* of the framework given in Chapter 5, and Section 5.4, schemas for the operations associated with the end-points (see Figure 6.3) are defined next.

End-points

As in the case of start points, the name of an operation associated to an end-point remains the same, but in lower case. In this work, the main purpose of those operations is to illustrate their roles in a specification, as suggested in Chapter 5.

The operation associated with the end-point $E1$ is described as:

$e1$

$\Xi netComTemp$

$id? : Identifier;\ resp! : Message$

$id? \in \text{dom}\ reqInvoices \land resp! = RequestSentToValidateInvoice$

$\lor$

$id? \in \text{dom}\ reqCustomers \land resp! = RequestSentToValidateCustomer$

$\lor$

$id? \in \text{dom}\ reqTransactions \land resp! = RequestSentToUpdateAccount$

$\lor$

$resp! = UnknownRequestId$

Depending on the type of a request sent over the network, the operation issues an appropriate message to the user. The variable $id?$ contains the identifier of the request sent.

At the end-point $E2$ (see Figure 6.3), the system must successfully have validated an invoice in order to pay a credit. The operation associated with this point removes from the abstract state space *netComTemp* all the information related to the completed request, and specifies a message that may be interpreted as a signal to allow the active scenario, to continue.

$e2$

$\Delta netComTemp$

$id? : Identifier;\ resp! : Message$

$reqCustomers' = id? \lhd reqCustomers \land reqResponses' = id? \lhd reqResponses$

$resp! = CustomerOk$

The input variable $id?$ contains the identifier of the request that was processed. The predicate part indicates that any information related to $id?$, is removed from the system.

The end-point $E3$ may be reached when both the invoice and the customer involved have been successfully validated, and the operation accepted (see the responsibility point named *AcceptOp* in Figure 6.2) by the beneficiary agency.

$e3$

$\Delta netComTemp$

$idi?, idc? : Identifier;\ resp! : Message$

$reqInvoices' = idi? \lhd reqInvoices \land reqResponses' = idi? \lhd reqResponses$

$reqCustomers' = idc? \lhd reqCustomers \land reqResponses' = idc? \lhd reqResponses$

$resp! = OperationAccepted$

The operation removes from the state space *netComTemp*, all information related to the input identifiers ($idi?$ and $idc?$), and specifies a message to report on its success.

In the light of *Guideline # 19*, in Chapter 5, the next section illustrates the implicit operation of the UCM abstract component, *Process* (see Figure 6.3).

The UCM component: Process

The UCM active component "process" plays an important role in the overall functioning of the system. It coordinates the execution of the operations in the system. To make such an operation visible, a control module was suggested in Chapter 5, page 87 based on the suggestion of van der Poll and Kotzé [91, 93]. To save space, the Z specification of this operation is not presented; its Object-Z specification is presented in Section 6.3.2.

In line with Guideline # 111 of the framework in Chapter 5, that suggests creating a meta-class for each UCM component, the following section defines such meta-classes.

Meta-classes

Due to the fact that the input UCM, contains only one abstract component (see Figure 6.3), only one meta-class is generated with the name, *ClsRequest*; a detailed description of this class is not discussed at this stage.

A number of cases were presented in (Chapter 3, Section 3.4.4) to illustrate how preconditions for partial operations may be calculated. To save space, preconditions will not be calculated in this chapter. However, to accommodate *Guideline # 21* of the framework suggested in Chapter 5, one example of such a calculation is presented in the following section, to illustrate the implementation of the framework.

Calculating preconditions

The precondition for the operation *reqCheckInvoice* is calculated, and the total operation derived from it.

The precondition is defined as:

Define *pre reqCheckInvoice* $\widehat{=}$ *preReqCheckInvoice* and the schema is:

$$
\begin{array}{|l}
\hline
\textit{preReqCheckInvoice} \\
netCompTemp;\ netInterface \\
id? : Identifier \\
\hline
\exists\, req : Request, reqToFoward' : Identifier \rightarrowtail\!\!\!\!\rightarrow Request \bullet \\
\qquad id? \in \operatorname{dom} reqInvoices \\
\qquad reqToFoward' = reqToFoward \cup \{id? \mapsto req\} \\
\hline
\end{array}
$$

The after state variable $reqToFoward'$ is existentially quantified. To simplify this schema, its predicate part is written in a textual form:

$(\exists\, req : Request, reqToFoward' : Identifier \rightarrowtail\!\!\!\!\rightarrow Request) \bullet$

1. $id? \in \operatorname{dom} reqInvoices$
2. $reqToFoward' = reqToFoward \cup \{id? \mapsto req\}$

By applying the One-point Rule, the existentially quantified variable $reqToFoward'$, is given an exact value. It may thus, be removed, to remain only with the condition in line #1. The predicate $id? \in \operatorname{dom} reqInvoices$, is therefore the precondition of the operation.

Negating the precondition yields the predicate $id? \notin \operatorname{dom} reqInvoices$ which is the precondition of the operation in case of an error, for which the schema is defined as followed:

$$
\begin{array}{|l}
\hline
\textit{reqCheckInvoiceFailed} \\
\Xi netComTemp \\
id? : Identifier;\ rep! : Message \\
\hline
id? \notin \operatorname{dom} reqInvoices \wedge resp! = Failed \\
\hline
\end{array}
$$

In line with the *Guideline # 22*, Chapter 5, the total operation is:,

$$reqCheckInvoice \mathrel{\widehat=} reqCheckInvoiceOk \vee reqCheckInvoiceFailed. \tag{6.8}$$

To complete the transformation of the input UCM, Object-Z classes (including meta-classes) are described next, in respect of *Guideline # 23* of the UCM transformation framework (see Chapter 5).

The Object-Z class schemas

In the previous sections, the Z specification of the plug-in (Figure 6.3) was presented. In this section, class schemas corresponding to the Object-Z transformation of the Z schemas are described. The transformation process requires each abstract state schema of Z, to become a class schema (Periyasamy and Mathew [68]). Such a class encapsulates the Z operation

schemas that operate on the state space. Wherever a meta-class is likely to encapsulate a state space, the class resulting from that same state space, is simply ignored. The two classes will be redundant since they encapsulate the same state, and the same set of operations that operate on it. As Z schemas were previously described, further explanations will be given only, where changes have occurred due, for example, to the transformation. First the class schema *ClsTimer*, generated from the abstract state schema *Timer*, is defined.

ClsTimer

$$\lceil(block, setup, respond, release, netFail, \textit{INIT})$$

$$opened : Boolean$$
$$waitingId : Identifier$$
$$maxtime : \mathbb{N}$$
$$opened = true \Rightarrow waitingid = \bot \land maxtime = 0$$

INIT

$$opened = true$$
$$waitingid = \bot$$
$$maxtime = 0$$

block

$$\Delta(opened, waitingId)$$
$$id? : Identifier$$
$$opened = true \land waitingid = id? \land opened' = false$$

release

$$\Delta(opened, waitindId, maxtime)$$
$$opened' = true \land waitingid = \bot \land maxtime' = 0$$

setup

$$\Delta(maxtime)$$
$$time? : \mathbb{N}$$
$$time?\ 0 \land maxtime' = time?$$

respond

$$id? : Identifier;\ resp?, resp! : Message$$
$$id? = waitingId \land resp! = resp?$$

netFail

$$resp! : Message$$
$$maxtime \leq 0 \land resp! = Failed$$

All the operations of this class are made accessible from the environment, especially to allow the UCM component, namely, process (see Figure 6.3) to control the activities of the Timer. The description of specific components and operations within the class remain unchanged compared to the Z version, except that in Object-Z, only one unnamed state schema is allowed in a class. The Δ operator, in an operation schema, applies to the list of variables that are changed by the operation, and the Ξ operator is ignored (see Duke and Rose [26], Smith [77]).

The next schema describes the class generated from the state schema *netInterface*. Because *netInterface* itself includes another state schema, the class for the internal schema, namely *ClsInterface*, is first presented.

ClsInterface

$reqToForward : Identifier \rightarrowtail\!\!\!\!\rightarrow Request$
$reqToReceive : Identifier \rightarrowtail\!\!\!\!\rightarrow Request$

INIT

$reqToForward' = \varnothing$
$reqToReceive' = \varnothing$

The Class *ClsInterface*, does not make any of its components accessible from outside. Its purpose is to facilitate the construction of the class *ClsNetInterface*, presented next.

ClsNetInterface

$\upharpoonright(\mathit{INIT}, send)$
ClsInterface

$(\forall\, id : Identifier;\ req : Request \bullet id \mapsto req \in reqToReceive)$
$(\exists_1\, net : ClsInterface \mid id \in \operatorname{dom}\ net.reqToFoward)$

send

$\Delta(reqToForward)$
$id? : Identifier;\ req? : Request$

$reqToFoward' = reqToFoward \cup \{id? \mapsto req?\}$

netReceive

$\Delta(reqReceive)$
$id? : Identifier;\ req? : Request$

$\mathbf{let}\ net : ClsNetInterface \mid id? \in \operatorname{dom} net.reqToFoward\ \bullet$
$\quad reqToReceive' = reqToReceive \cup \{id? \mapsto req?\}$

The operations *send* and *netReceive* are accessible externally. The class extends *ClsInterface* to include a constraint on the set of requests received from other companies. The operation *netReceive* updates the component *reqToReceive* with a new request that arrives from another agency.

The class *ClsNetComTemp* is:

ClsNetComTemp

$$\lceil(reqInvoices, reqCustomers, reqTransactions, reqResponses, reqCheckInvoices, reqCheckCustomer, reqUpdateAccount, Response)$$

$$forward \mathrel{\widehat=} [network? : ClsNetInterface] \bullet network.send$$

$$reqInvoices : Identifier \pfun Invoice \times Address$$
$$reqCustomers : Identifier \pfun Customer \times Address$$
$$reqTransactions : Identifier \pfun Customer \times Money \times Address$$
$$reqResponses : Identifier \fun Message$$

$$\mathrm{dom}\ reqInvoices \cap \mathrm{dom}\ reqCustomers = \varnothing$$
$$\mathrm{dom}\ reqInvoices \cap \mathrm{dom}\ reqTransactions = \varnothing$$
$$\mathrm{dom}\ reqCustomers \cap \mathrm{dom}\ reqTransactions = \varnothing$$
$$\mathrm{dom}\ reqResponses \subseteq \mathrm{dom}\ reqInvoices \cup \mathrm{dom}\ reqCustomers \cup \mathrm{dom}\ reqTransactions$$

INIT

$$reqInvoices = \varnothing \land reqCustomers = \varnothing$$
$$reqTransactions = \varnothing \land reqResponses = \varnothing$$

reqCheckInvoice

$$id? : Identifier$$

$$id? \in \mathrm{dom}\ reqInvoices$$
$$\exists\, req : Request \bullet forward(id?, req)$$

reqCheckCustomer

$$id? : Identifier$$

$$id? \in \mathrm{dom}\ reqCustomers$$
$$\exists\, req : Request \bullet foward(id?, req)$$

reqUpdateAccount

$$id? : Identifier$$

$$id? \in \mathrm{dom}\ reqTransactions$$
$$\exists\, Request \bullet forward(id?, req)$$

This class derives from the Z state space *netComTemp*. It defines the operation *forward* in terms of the *send* operation of *ClsNetInterface*, making it usable within the class *ClsNetComTemp*. The definition of *forward* requires the environment to provide an object of *ClsNetInterface*, from which the operation *send* is promoted (more insight on promoting Object-Z operations may be found in the books by Duke and Rose [26] and Dunne [28]). Other operations of the class use the promoted operation to send requests over the network. Those operations do not include the Δ operator, because the state of the network interface is not directly accessible from outside, and is therefore updated through the promoted operation.

Next the meta-class, *ClsRequest*, previously introduced, is defined more comprehensively. It inherits variables from the class of variables *ClsGlobalVariables*, and properties from two other classes: *ClsTimer* and *ClsNetComTemp*. The two classes *StartPoints* and *EndPoints* are also included even though they are not described in detail. The class *StartPoints* encapsulates all the Z schemas to the start-points, and the class EndPoints groups all the Z schemas associated to all the end-points. The class *ClsRequest* defines by promotion, the operations *endCheckInvoice* and *endCheckCustomer* as well as three blocks of composite operations that are explained in subsequent paragraphs.

ClsRequest

$\lceil(s1, s2, s3, s4)$
ClsGlobalVariables
ClsTimer
ClsNetComTemp
StartPoints
EndPoints

$$endCheckInvoice \mathrel{\widehat=} [\exists\, id \in (\mathrm{dom}\, reqCheckCustomer) \cap (\mathrm{dom}\, reqResponses) \mid idScenario(id) = idScenario(id1?)] \bullet e3(id1?, id)$$

$$endCheckCustomer \mathrel{\widehat=} [idScenario(id2?) = scenePayCredit] \bullet e2(id2?) \lor [\exists\, id \in \mathrm{dom}\, reqCheckInvoice \cap \mathrm{dom}\, reqResponses \mid idScenario(id) = idScenario(id1?)] \bullet e3(id, id2?)$$

Activity #1

$$[id1? : Identifier, inv? : Invoice, ad? : Address] \bullet s1 \fatsemi reqCheckInvoice(id1?) \fatsemi e1(id1?) \land (block \land setup) \fatsemi ((s4 \fatsemi respond) \lor (netFail \fatsemi e123b)) \land release) \fatsemi endCheckInvoice$$

Activity #2

$[id2? : Identifier, cust? : Customer, ad? : Address] \bullet s2 \fatsemi_9$
$\{(reqCheckCustomer(id2?) \fatsemi_9 e1(id2?)) \wedge (block \wedge setup)$
$\circ_9$
$((s4 \fatsemi_9 respond) \vee (netFail \fatsemi_9 e123b)) \wedge release\} \fatsemi_9 endCheckCustomer$

Activity #3

$[id3? : Identifier, cust? : Customer, amnt? : Money, ad? : Address] \bullet s3 \circ_9$
$reqUpdateAccount \fatsemi_9 e1$

The operation *endCheckInvoice* terminates the validation process of an invoice. It uses the scope enrichment operator $\bullet$ (see Chapter 2) to promote the operation $e3$. When the condition imposed by the AND-join element, described above, is reinforced, at this stage, the customer was thusable to be successfully validated. The system determines the identifier *id*, which is used by the operation $e3$, to terminate the appropriate scenario. The other parameter $id1?$, is the identifier of the request that is used (in the composite operation labeled **Activity #1**) to validate the corresponding invoice. The condition $idScenario(id) = idScenario(id1?)$ ensures that the two requests referenced by *id* and $id1?$ are linked to the same scenario. This operation aims to shorten the expression of **Activity #1**

Similarly to *endCheckInvoice*, the operation *endCheckCustomer* terminates the validation process of a customer. It is a composite operation that acts according to the scenario in execution. It either promotes the operation $e2$ when the customer is paying a credit, or acts like the operation *endCheckInvoice*, when returning or replacing a purchased item. To reinforce the effect of the AND-join connector, it ensures that both the customer and invoice involved are successfully validated. The identifier $id2?$ was provided by the environment when triggering the start-point $S2$ (see block of **Activity #2**), and identifies the request used to validate the customer. This operation aims to shorten the expression of **Activity #2**.

The three main activities of this sub-system are the following:

1. **Activity #1**
   This composite operation describes the reaction of the system, when the start-point $S1$ is triggered to initiate the validation of an invoice.

2. **Activity #2**
   This operation describes the sequence of activities performed by the system when the

start-point $S2$ is triggered to initiate the validation of a customer.

3. **Activity #3**
   This schema expression defines the sequence of activities that the system performs in reaction to the triggering of the start-point $S3$, to update a customer's account.

In the definition of the class *ClsRequest*, the labels listed above are merely text comments used to reference the expression of the composite operations placed below them. The paragraph that follows explains **Activity #1** so that the two others may be similarly understood.

Recall from the previous discussion on partial operations, that the reason to trigger the start-point $S1$, is to initiate the validation of an invoice. For such a triggering event to succeed, the environment provides appropriate information to render the operation $s1$ applicable. In the expression of the composite operation, the scope enrichment operator ($\bullet$) is introduced to make it possible to indicate that a request identifier $id1?$, the invoice to be validated $inv?$, and the address $ad?$ of the agency to which the request is to be forwarded, are the information required from the environment, to enable start the validation process. The activities of the system involve:

1. Handling the triggering event with the operation $s1$,

2. Requesting, with the identifier $id1$, the validation of the input invoice, block and set-up the timer, with an appropriate maximum waiting time as indicated by the expression below:

   $$(reqCheckInvoice(id1?) \fatsemi e1(id1?)) \wedge (block \wedge setup)$$

   At the level of this sub-system, the sending operation terminates at the end-point $E1$ (see the UCM in Figure 6.3), hence, the operation $e1$ associated with the end-point immediately follows the operation *send*.

3. Waiting for the start-point $S4$ (Figure 6.3) to be triggered, indicating the availability of a response, therefore the operation *Respond* is performed.

   $$s4 \fatsemi respond$$

   Or waiting for a timeout event to occur indicating a failure; when the process fails, the system reacts by performing *netFail* and the progression of the scenario is terminated at $E123b$ with the operation $e123b$ associated.

   $$netFail \fatsemi e123b$$

   At that point, either a response is received, or a timeout has occurred, the timer is released and the whole validation process is terminated (see earlier discussion on the operation *enCheckInvoice*).

As discussed earlier in this chapter (see Section 6.2), the plug-in for which the Object-Z transformation has just been completed in this section, has the responsibility to forward requests from one agency, to another, using the network support. The other two plug-ins in figures 6.4 and 6.5, respectively, handle those requests from the network aiming to validate invoices and customers. Thus, continuing with the Z and Object-Z specification of the input UCM (Figure 6.1), the following two sections focus on the remaining two plug-ins. Due to the reason that neither of the two sub-maps include new UCM elements that have not yet been discussed previously, for brevity, some steps of the transformation process are omitted. Only the resulting Object-Z classes are discussed here.

6.3.3 Applying the framework to the Plug-in to validate invoices (Figure 6.4)

As mentioned previously, the Z transformation of the sub-map in Figure 6.4 is not presented.

This section describes two class schemas: *ClsLocalSales* and *ClsCheckInvoices* that represents the Object-Z specification resulting from the transformation of the diagram in Figure 6.4.

1- The class *ClsLocalSales* describes objects that interface with the local sales system (Figure 3.2 shows the sub-system layout). It makes accessible to the environment the functions *findInvoice* and *findAccount*, and is not included in either of the two plug-ins. Instead, it provides two functions to help them query a local sales system of an agency when validating an invoice or a customer.

ClsLocalSales

$\lceil(findInvoice, findAccount)$

$$invoices : Invoice \pfun Customer$$
$$customers : Customer \pfun Account$$
$$\dom invoices \subseteq \ran customers$$

INIT

$$invoices = \varnothing \land customers = \varnothing$$

findInvoice

$$inv? : Invoice;\ resp! : \mathbb{B}$$
$$inv? \in \dom invoices \land resp! = true$$
$$\lor$$
$$resp! = false$$

$$
\begin{array}{|l}
\textit{findAccount} \\
\hline
\textit{cust}? : \textit{Customer};\ \textit{resp}! : \mathbb{B} \\
\hline
\textit{cust}? \in \operatorname{dom} \textit{customers} \land \textit{resp}! = \textit{true} \\
\lor \\
\textit{resp}! = \textit{false}
\end{array}
$$

The operation *findInvoice* checks the availability of an input invoice in the local sales system. A boolean value is used to specify a message returned to the user to indicate whether the invoice is in the system, or not. The operation *findCustomer* acts in a similar way as *findInvoice* relative to customers.

2- The class *ClsCheckInvoice* checks invoices responding to remote requests. Its state defines a set of requests. The variable $out12$ indicates whether or not the validation has succeeded and hence, guides further decisions to be taken. An object of the class may be initialised from the system environment, with the operation $\textsc{Init}$.

$$
\begin{array}{|l}
\textit{ClsCheckInvoice} \\
\hline
\upharpoonright(\textsc{Init}, s1, out12) \\
\\
\begin{array}{|l}
\hline
out12 : \mathbb{B} \\
\textit{requests} : \textit{Identifier} \rightarrowtail\!\!\!\!\twoheadrightarrow \textit{Invoice} \\
\hline
\end{array} \\
\\
\begin{array}{|l}
\textsc{Init} \\
\hline
\textit{requests} = \varnothing \land out12 = \bot \\
\hline
\end{array} \\
\\
\begin{array}{|l}
s1 \\
\hline
\Delta(\textit{requests}) \\
\textit{id}? : \textit{Identifier};\ \textit{inv}? : \textit{Invoice} \\
\hline
\textit{requests}' = \textit{requests} \cup \{\textit{id}? \mapsto \textit{inv}?\} \\
\hline
\end{array} \\
\\
\begin{array}{|l}
e11 \\
\hline
\Delta(\textit{requests}) \\
\textit{id}? : \textit{Identifier};\ \textit{resp}! : \textit{Message} \\
\hline
\textit{requests}' = \{\textit{id}?\} \lhd \textit{requests} \land \textit{resp}! = \textit{UnknownInvoice} \\
\hline
\end{array} \\
\\
\begin{array}{|l}
e12 \\
\hline
\Delta(\textit{requests}) \\
\textit{id}? : \textit{Identifier} \\
\hline
\textit{requests}' = \{\textit{id}?\} \lhd \textit{requests} \\
\hline
\end{array}
\end{array}
$$

$$ChkInv \mathrel{\widehat=} [view? : ClsLocalSales] \bullet view?.findInvoice(inv, out12)$$

$$[\exists(id, inv) \in requests \mid requests \neq \varnothing] \bullet ChkInv \fatsemi$$
$$[\neg\, out12] \land e11$$
$$\lor$$
$$[out12] \land e12$$

The operation $s1$ is awaken to keep a new request in the system each time the start-point $S1$, is triggered with an appropriate identifier associated to the request. The two operations $e11$ and $e12$, respectively associated with the end-points $E11$ and $E12$ of the plug-in (Figure 6.4), are defined to terminate the validation process of an invoice. The class also uses the scope enrichment operator to define the operation *ChkInv*. The environment provides the object *view*? of the class *ClsLocalSales*, for which, the operation *findInvoice* is put forward to be used locally as *ChkInv*, to validate invoices. The last expression describes the reaction of the system against the set of incoming requests. One request is selected from the list, and checked with the operation *ChkInv*. Depending on whether the validation has failed $[\neg\, Out12]$ or not, the system continues to the end-point $E11$ or $E12$.

Similarly, the next section presents the single class that results from the Object-Z transformation of the plug-in to validate, a customer.

6.3.4 Applying the framework to the Plug-ins to validate customers (Figure 6.5)

The only Object-Z class schema resulting from the diagram in Figure 6.5 is the class *ClsCheckCustomer*. The definition of the class *ClsCheckCustomer* is very similar to that of *ClsCheckInvoice*, in the sense that both include in their state, a set of incoming requests and inherit operations from the class *ClsLocalSales* to query the local sales system of an agency. This classes validates customers and forwards the requests to update a customer's account to the end-point $E23$ (see Figure 6.5). The component $out345$ indicates the result of actions taken within the class, and aims to guide further decisions. The possible values of the component are:

1. $out345 = 3$: indicates that the validation of the current customer has failed, and the end-point $E21$ is reached;

2. $out345 = 4$: indicates that the validation of the current customer has succeeded and the end-point $E22$ is reached, or

3. $out345 = 5$: indicates that the current request is to update the customer's account, and the end-point $E23$ is reached.

The end-points $E21$, $E22$ and $E23$ are those on the UCM of Figure 6.5. The operation $s2$ adds a new request into the set of pending requests each time the start point $S1$ is triggered.

ClsCheckCustomer

$\upharpoonright(\text{INIT}, s2, out345)$

$out345 : \mathbb{N}$
$requests : Identifier \rightarrowtail\!\!\!\!\!\rightarrow Customer$

$out345 \neq \perp \Rightarrow out345 \in \{3, 4, 5\}$

INIT

$requests = \varnothing \wedge out345 = \perp$

s2

$\Delta(requests)$
$id? : Identifier, cust? : Customer$

$requests' = requests \cup \{id? \mapsto cust?\}$

e21

$\Delta(requests)$
$id? : Identifier;\ rep! : Message$

$requests' = \{id?\} \lhd requests \wedge rep! = UnknownCustomer \wedge out345 = 5$

e22

$\Delta(requests)$
$id? : Identifier$

$requests' = \{id?\} \lhd requests \wedge out345 = 3$

e23

$\Delta(requests)$
$id? : Identifier$

$requests' = \{id?\} \lhd requests \wedge out345 = 4$

$ChkCust \mathrel{\widehat=} [view? : ClsLocalSales;\ resp! : \mathbb{B}] \bullet view?.FindAccount(cust, resp!)$

$[\exists (id, cust) \in requests;\ op? \in \{check, update\} \mid request \neq \varnothing] \bullet$
$\quad \textbf{if } op? = check \textbf{ then } ChkCust \,;\, (([\neg\, resp!] \wedge e21) \vee ([resp!] \wedge e22))$
$\quad \vee$
$\quad \textbf{if } op? = update \textbf{ then } e23$

The operations $e21$, $e22$ and $e23$ are, respectively, associated to the end-points $E21$, $E22$ and $E23$, and are activated each time the end-point is reached. The class also defines *ChkCust* to promote the function *findAccount* of *ClsLocaleSales*. The last expression uses the scope enrichment operator ($\bullet$) to describe the sequence of actions taken by the plug-in to react whenever there are pending requests. The first expression of the operator $[\exists(id, cust) \in requests;\ op? \in \{check, update\} \mid request \neq \varnothing]$ indicates that the system selects a pending request, and the environment provides a value to the input parameter $op?$, to indicate the nature of the action needed. The expression at the right side of the operator, indicates the sequence of actions taken by the system depending on whether the request is to validate a customer ($op? = check$), or to update a customer's account ($op? = update$).

At this point, the transformation of the three plug-ins resulting from the break-down of the initial input UCM (see Figure 6.6) is completed. In the next section, the Object-Z transformation of the stubbed map, namely, main map is presented .

6.3.5 Applying the framework to the main UCM

Traversing the main UCM in Figure 6.2 from left to right, reveals that the map includes hierarchical structured abstract components and stubs. Those UCM elements taken together with the sub-maps or plug-ins, (treated in the previous sections), model the envisioned system. Figure 6.7 illustrates the hierarchical structuring of those components.

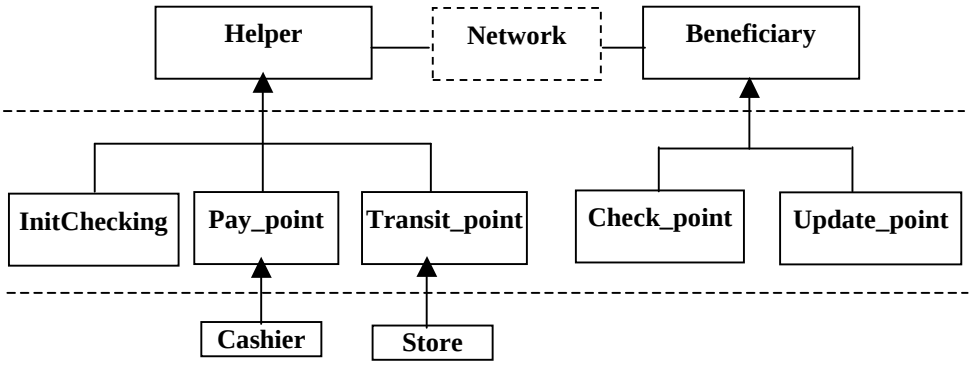


Figure 6.7: The hierarchical structuring of components in the stubbed UCM

A component at a higher level of the hierarchy, serves as a container for those at the next lower level attached to it by means of directional arrows. As suggested in the framework (Section 5.4, Chapter 5), during the transformation process, a bottom-up strategy is adopted. To accommodate the *Guidelines # 11 and # 12* of the framework, components at the lowest

level are considered first, followed by those at the next higher level, until the ones at the highest level are considered.

The structuring (in Figure 6.7) of the main UCM (in Figure 6.2), reveals the two major roles played by the system in an agency:

(a) ***Helper role***
The system plays a "helper" role whenever the agency, in which it is operating, assists a customer of another agency, aiming to achieve one of the three user requirements listed in the case study (see Chapter 3). In that case, the operations needed are provided by the UCM components encompassed by the abstract component, namely, **Helper** (see Figure 6.2), or connected to it, in Figure 6.7, (by means of the arrows).

(b) ***Beneficiary role***
The system plays a "Benefiary" role whenever the agency, in which the system is operating, collaborates remotely with the agency that is helping its customer. In that case, the operations needed are provided by the UCM components encapsulated in the component, namely, **Beneficiary** (see Figure 6.7), or connected to it, in Figure 6.7), (by means of the arrows).

The communication between a Helper and a Beneficiary is conducted via the services of a network component, which is assumed to be external to the system. However, the communication interfaces are part of the system, since they are necessary to enable the use of the network. Each UCM component contained in one of the two abstract components at the highest level of hierarchy in Figure 6.7, may request the services of the network. For example, the component "Helper" uses the UCM stub element to control and monitor network connections. The UCM paths joining the stub and other abstract components (see Figure 6.2) indicate a network communication involving those components.

As discussed in the previous sections, the stubs included in this main map (in Figure 6.2) are:

- A static stub, to which is bound the Plug-in, to forward requests (Figure 6.3), and
- A dynamic stub, to which are associated the plug-in to validate invoices, (Figure 6.4) and the plug-in to validate customers (Figure 6.5).

A stub in a UCM represents a place where the service of a sub-map (plug-in) is needed. In Chapter 5, *Guideline # 14* of the UCM transformation framework, as well as Section 5.3 (relating the static and dynamic stub concepts of UCM to Z schemas) were proposed to assist

the Z specification of a binding relationship, between a stub and its plug-in(s). For brevity, the step-by-step Z transformation of the main map is not presented in detail. However, Table 6.1 is constructed to indicate the list of the Z abstract state schemas generated. For each schema, the UCM element, from which the schema was derived, is also shown. Similarly, Table 6.2 is constructed to show the list of the Z partial operations generated.

The step-by-step application of the transformation process was implemented in the previous sections when generating the Z and Object-Z versions of the sub-maps in figures 6.3, 6.4 and 6.5. Therefore, repeating the same details for the main map or similar UCM elements will certainly increase the size of this dissertation, but may not provide any new insight. Hence, for the Z and Object-Z specification of the main UCM, only results are presented (details pertaining to the transformation process are not shown).

The next section presents in a tabular form, the list of the abstract state schemas generated.

Summary of Z state schemas

As mentioned above, the list of state schemas resulting from the Z transformation of the stubbed UCM is presented. Each schema is associated with the UCM elements from which the Z schema was generated. The type of the UCM element is also presented, to show that the Z elements were generated on the strength of the proposed guidelines (Section 5.4, Chapter 5). An abstract state schema associated with a UCM element, encapsulates the properties of the system that is under the control of the UCM element.

State schema	**UCM element**	**Type of Element**
stateInitChecking	*InitChecking*	process component
stateCashier	*Cashier*	object component
statePayPoint	*Pay_point*	team component
stateInStore	*Store*	object component
stateTransitPoint	*Transit_point*	team component
stateHelper	*Helper*	team component
stateCheckPoint	*Check_point*	team component
stateUpdatePoint	*Update_point*	team component
stateBeneficiary	*Beneficiary*	team component

Table 6.1: List of the Z abstract state schemas

In accordance with *Guideline # 12* of the framework (in Section 5.4, Chapter 5), each state

schema derived from a UCM team component that contains other components, includes the schemas generated from the integral components. For example, the schema named *statePayPoint*, includes the state schema *stateCashier*, which is in turn encapsulated in the *stateHelper*, that also includes *stateInitChecking*, and *StateTransitPoint*.

The list of meta-classes is presented next.

Meta-classes generated from Main UCM

The list of the meta-classes pertaining to the idea of creating a meta-class for each UCM active component (see *Guideline # 111*, Chapter 5) are listed below:

- *ClsInitChecking*, derived from the component *InitChecking*
- *ClsPayPoint*, derived from the component *Pay_point*
- *ClsTransitPoint*, derived from the component *Transit_point*
- *ClsCheckPoint*, derived from the component *Check_point*
- *ClsUpdatePoint*, derived from the component *Update_point*
- *ClsHelper*, derived from the component *Helper* and
- *ClsBeneficiary*, derived from the component *Beneficiary*.

Meta-classes are not associated to Object components, as those componens do not have control over the tasks they perform. Detailed descriptions of meta-classes are presented later in this chapter.

Next, the list of the Z partial operations generated from the stubbed UCM is shown in a tabular form.

Summary of Z partial operations

Following the suggestion of Potter et al. [70] wherein the idea was put forward that:

"It is useful to give a summary of what is discovered in a table, showing for each operation its inputs, outputs and preconditions."

A non-conventional way of presentation follows. Without intending to change the standard way of presenting Z partial operations, in Table 6.2, in place of preconditions, a prose

text description to indicate the purpose of the operation is shown. Formal definitions of some of the operations are discussed in the forthcoming section, where Object-Z classes that encapsulates them, are presented.

Operation	Inputs and Outputs	Purpose
s1	$item? : Item$ $invoice? : Invoice$ $customer? : Customer$ $agency? : Agency$	Starts a scenario to return an item
s2	$item? : Item$ $invoice? : Invoice$ $customer? : Customer$ $agency? : Agency$	Starts a scenario to replace an item
s3	$customer? : Customer$ $agency? : Agency$	Starts a scenario to pay a credit
initCheckInvoice	$item? : Item$ $invoice? : Invoice$ $agency? : Agency$ $resp! : Identifier \times Invoice \times Address$	Prepares a request to be sent over the network to check the validity of an invoice
initCheckCustomer	$cust? : Customer$ $agency? : Agency$ $resp! : Identifier \times Invoice \times Address$	Prepares a request to be sent over the network to check the validity of a customer
payCredit	$cust? : Customer$ $amnt? : Money$ $agency? : Agency$ $date? : Date$	Handles a customer's payment at a cashier
allocTransaction	$cust? : Customer$ $agency? : Agency$ $date? : Date$ $resp! : Identifier \times Customer \times Money \times Address$	Transfers a payment into a specific account
storeItem	$cust? : Customer$ $item? : Item$ $inv? : Invoice$ $date? : Date$	Keeps a returned item in a store before shipping

shipItem	*item*? : *Item* *date*? : *Date*	Ships a returned item to the provider
acceptOp	*inv*? : *Invoice* *resp*! : $\mathbb{B}$ *item*? : *Item*	Provider allows a return or replace operation to continue or not
receivItem	*item*? : *Item* *inv*? : *Invoice* *from*? : *Agencyid*	Provider receives a shipped item
updateCustAcc	*amount*? : *Money* *custAcc*? : *Account* *resp*! : *Message*	Provider updates a customer's account to reflect a payment or the value of a returned item
$e12a$	*inv*? : *Invoice* *resp*! : *Message*	Terminates a scenario when the invoice is not valid
$e12b$	*inv*? : *Invoice* *resp*! : *Message*	Terminates a scenario when provider of a returned item does not allow the operation
$e123a$	*cust* : *Customer* *resp*! : *Message*	Terminates a scenario when customer does not have valid account with the provider
deliverItm	*item*? : *Item* *customer*? : *Customer*	Deliver a new Item to Customer to replace the returned one
refundCustomer	*amount*? : *Money* *customer*? : *Customer*	Refund a customer for a returned Item
$e1$	*inv*? : *Invoice* *item*? : *Item* *resp*! : *Message*	Successfully terminates a scenario after customer is refunded for a returned item
$e2$	*inv*? : *Invoice* *item*? : *Item* *resp*! : *Message*	Successfully terminates a scenario after a returned item is replaced by the provider
$e3$	*amount*? : *Money* *cust*? : *Customer* *resp*! : *Message*	Successfully terminates a scenario to pay credit

Table 6.2: Partial operations for the main UCM in fig.6.2

As suggested by the UCM transformation framework (see *Guideline # 23*, Chapter 5), the

following Section aims to complete the specification of Object-Z class schemas, generated earlier in this section.

The Object-Z Class schemas

By applying the UCM transformation framework, proposed in Chapter 5, to the stubbed UCM (in Figure 6.2), Object-Z class schemas are generated. Some derive from meta-classes directly associated to UCM abstract components that may have control over the tasks they perform; others are derived from the transformation of the Z abstract state spaces obtained from the input UCM (see Chapter 4). Such a transformation was illustrated in Section 6.3.2, during the transformation process of the Plug-in to forward requests. The list of the generated classes is:

#	**Class name**	**Derived from**
01	*ClsInitChecking*	meta-class
02	*ClsCashier*	*stateCashier*
03	*ClsPayPoint*	meta-class
04	*ClsInStore*	*stateStore*
05	*ClsTransitPoint*	meta-class
06	*ClsHelper*	meta-class
07	*ClsCheckPoint*	meta-class
08	*ClsUpdatePoint*	meta-class
09	*ClsBeneficiary*	meta-class
10	*ClsMainStartPoints*	*stateHelper*
11	*ClsMainEndPoints*	*stateHelper*

Table 6.3: List of OZ classes for the stubbed UCM

The task of the two classes *ClsHelper* and *ClsBeneficiary* generated, respectively, from the UCM abstract components *Helper* and *Beneficiary* (see figures 6.7 and 6.2), is to monitor (through the mechanism of inheritance) the activities of the other classes generated from the UCM components, or elements that they encapsulate. A complete description of those two classes may be valuable to show how the UCM transformation framework, (which is one of the main contributions of this dissertation), can be applied to connect static and dynamic stubs to the corresponding sub-maps. Note that a complete development of the other classes may not be so important, since similar classes have already been developed in

previous sections.

In Figure 6.2, by following a UCM path from the start-point $S1$ or $S2$ to the end-point $E1$ or $E2$, it appears that a complete scenario, for returning or replacing a purchased item, can be traced, by considering the classes *ClsInitChecking*, *ClsTransitPoint*, *ClsHelper*, *ClsCheckPoint*, *ClsUpdatePoint*, and *ClsBeneficiary*. For the reasons stated in the above paragraph, only the classes *ClsHelper* and *ClsBeneficiary* are fully described. The other four classes are partially described: the predicate part of some operations are not presented here. The rest of the classes listed in Table 6.3 are not presented.

Next, the Class *ClsInitChecking* is defined.

The class *ClsInitChecking*

The purpose of this class is to prepare requests to be sent over the network to check an invoice or a customer. The class *ClsInitChecking* is the comprehensive version of the meta-class, obtained from the abstract component named *InitChecking*. It inherits variables from the class *ClsGlobalVariables* (see Section 6.3.1) and properties from the class *ClsMainStartPoints*, (which is not defined here, as mentioned above).

This class encapsulates two operations, *initCheckInvoice* and *initCheckCustomer*, to initialise a request to check respectively, an invoice or a customer, and generate an identifier for the request. Its schema is shown next followed, by further explanations.

ClsInitChecking

$\lceil(\textit{INIT}, \textit{initCheckInvoice}, \textit{initCheckCustomer})$

ClsGlobalVariables

ClsMainStartPoints

initCheckInvoice

$item? : Item;\ inv? : Invoice;\ agency? : Agency$

$resp! : Identifier \times Invoice \times Address$

$(item, inv?, cmpy?) \in \operatorname{dom} items$

$(\exists\, id : Identifier) \bullet resp! = (id, inv?, addressOf(agency?))$

initCheckCustomer

$cust? : Customer;\ agency? : Agency$

$resp! : Identifier \times Customer \times Address$

$(cust?, agency?) \in \operatorname{dom} customers$

$(\exists\, id : Identifier) \bullet resp! = (id, cust?, addressOf(agency?))$

This class keeps temporary information on a scenario, as long as the scenario is in progress.

The component that holds temporary information is inherited from the class *ClsMainStartPoints* and comprises three records:

$scenarios : \mathbb{F}\, Scenario$, which defines the finite set of active scenarios;

$items : Item \times Invoice \times Company \nrightarrow Scenario$, which defines the set of items to be returned or to be replaced, depending on the scenario under consideration.
Each element of the set is composed of: an item; an invoice (which is the proof of purchase of the item); and the agency where the item was purchased;

$customers : Customer \times Agency \nrightarrow Scenario$ defines the set of customers whose requests are being processed.

A record is maintained in *items* or *customers* only as long as the scenario for which it was created remains active. Initially, the component is assumed to be empty.

The precondition for the operation *initCheckInvoice* requires the triple elements formed by the values of the input variables *item*?, *inv*? and *agency*? to be mapped, in the temp file, to a scenario in progress. The operation generates an identifier (that is output) together with *inv*?, and the address of the target agency. The function *addressOf* is inherited from *ClsGlobalVariables*.

The class *ClsTransitPoint*

The class *ClsTransitPoint* provides a complete version of the meta-class generated from the UCM team component *Transit_Point*, (that includes the UCM object element *Store*). It temporarily keeps (in *Store*) items received from customers, and ships them to their providers. It inclusively inherits the list of stored items, from the class *ClsInStore*, and additionally contains the variable *shippedItems*, to record the set of items taken from the store and shipped to the appropriate agencies. It is assumed that the system starts operating with an empty list of items.

ClsTransitPoint

$\lceil(shippdItems, \textsc{Init}, shipItem)$

ClsInStore

$shippedItems : Item \pfun Date$

$\mathrm{dom}\ shippedItems \subseteq \mathrm{dom}(\mathrm{dom}\ storedItem)$

Init

$shippedItems = \varnothing$

shipItem

$\Delta(shippedItems)$
$item? : Item;\ date? : Date$

$item? \in \mathrm{dom}(\mathrm{dom}(\mathrm{dom}\ storeItems))$
$shippedItems' = shippedItems \cup \{item? \mapsto date?\}$

The class inherits the operation *storeItem* from the class *ClsInStore*, and defines the operation *shipItem*, to perform the activity of transferring an item, from the store to the provider of the item.

Next is the class that defines the activities of the sub-system, namele, *Helper*.

The class *Helper*

The class *ClsHelper*, is an update of the meta-class derived from the UCM component named *Helper*, which normally represents a sub-system. The class specifies the chain of activities performed by the sub-system for scenarios that are in progress.

ClsHelper

$\lceil(\textsc{Init}, startFromS1, startFromS2, startFromS3)$

$plugin : ClsRequest$

Init

$plugin.\textsc{Init}$

$IN1 \mathrel{\widehat=} [id? : Identifier;\ inv? : Invoice;\ addr? : Address] \bullet plugin.s1$
$IN2 \mathrel{\widehat=} [id? : Identifier;\ cust? : Customer;\ addr? : Address] \bullet plugin.s2$
$IN3 \mathrel{\widehat=} [id? : Identifier;\ cust? : Customer;\ cash? : Money;$
$addr? : Address] \bullet plugin.s3$

$$
\begin{array}{l}
startFromS1 \mathrel{\widehat=} [item? : Item;\ inv? : Invoice;\ cust? : Customer; \\
\quad agency? : Agency;\ checker : ClsInitChecking] \bullet \\
\qquad (checker.s1 \fatsemi ((checker.initCheckInvoice \fatsemi IN1) \land \\
\qquad (checker.initCheckCustomer \fatsemi IN2)) \\
\qquad \fatsemi \\
\qquad [store? : ClsTransitPoint] \bullet (store?.storeItem \fatsemi store?.shipItem) \\
startFromS2 \mathrel{\widehat=} [item? : Item;\ inv? : Invoice;\ cust? : Customer; \\
\quad agency? : Agency;\ checker : ClsInitChecking] \bullet \\
\qquad (checker.s2 \fatsemi ((checker.initCheckInvoice \fatsemi IN1) \land \\
\qquad (checker.initCheckCustomer \fatsemi IN2)) \\
\qquad \fatsemi \\
\qquad [store? : ClsTransitPoint] \bullet (store?.storeItem \fatsemi store?.shipItem) \\
startFromS3 \mathrel{\widehat=} [cust? : Customer;\ agency? : Agency; \\
\quad checker : ClsInitChecking] \bullet \\
\qquad (checker.s3 \fatsemi (checker.initCheckCustomer \fatsemi IN2) \fatsemi \\
\qquad [payer? : ClsPayPoint] \bullet (payer?.payCredit \land \\
\qquad\quad payer?.allocTransaction) \fatsemi \\
\qquad payer.initUpdateCustomerAcc \fatsemi IN3)
\end{array}
$$

An object of this class has the important role of coordinating the sequence of operations that may be performed within an agency, when acting as a *Helper*. It uses the variable *plugin* (an object of the class *ClsRequest*), to inherit properties and methods from the class *ClsRequest* (see Section 6.3.2), generated from the UCM sub-map (in Figure 6.3), connected to the static stub.

The operations $IN1$, $IN2$ and $IN3$ are specified to define input points to *plugin*. For each input variable in the square brackets, each of those points is activated whenever a value is provided (e.g. from the system environment). When those values are provided, for example, for the $IN1$, the start-point $S1$ of the plugin is triggered (*plugin*.$s1$), and the sequence of activities within the plugin (see Section 6.3.2) that follows, are performed.

Each of the three operations *startFromS*1, *startFromS*2, and *startFromS*3 specifies a sequence of activities performed whenever any of the start-points ($S1$, $S2$, or $S3$) (see Figure 6.2), is triggered. For example, a scenario to return a purchased item, may commence if a value is provided for each of the input variables in the square brackets, and an object of the class *ClsInitChecking* is created. In that case, the operation $s1$ (*Checker*.$s1$), handles the triggering event, to temporarily conserve the values for the input variables, for further use. A request to check the invoice is prepared (*Checker.initCheckInvoice*) and submitted via the input point $IN1$. Concurrently, a request to check the customer is prepared

(*Checker.initCheckCustomer*), and submitted via the input point *IN*2. As shown in the above paragraph, *IN*1 or *IN*2 connects to the *plugin*, that controls the communication, through the network. An object of the class *ClsTransitPoint* is required from, (or created by), the system environment to temporarily keep (*store*?.*storeItem*) the input item in a store, before shipping it (*store*?.*shipItem*) to the provider.

The construction of this class requires the specification of the classes *ClsRequest*, (seen in Section 6.3.2), *ClsInitChecking*, *ClsTransitPoint*, and *ClsPayPoint*, which were all generated from UCM elements, included in the team component *Helper* (as shown in figures 6.2 and 6.7). Such a construction illustrates the use of the bottom-up strategy adopted in the transformation framework (see Chapter 5, Section 5.4). Similar reasoning is followed to construct the class *ClsBeneficiary*, generated from the UCM team component *Beneficiary* (see Figure 6.7).

The class *ClsCheckPoint*

The Class *ClsCheckPoint* specifies the activity to collect and process incoming requests from the network.

ClsCheckPoint

$\lceil(\textit{INIT})$

ClsGlobalVariables

$$
\begin{array}{l}
\textit{collectedReq} : \textit{Identifier} \rightarrowtail\!\!\!\!\!\!\rightarrow \textit{Request} \\
\textit{plugin4Inv} : \textit{ClsCheckInvoice} \\
\textit{plugin4Cust} : \textit{ClsCheckCustomer} \\
\hline
\textbf{let}\ \textit{listReq} == \operatorname{ran} \textit{collectedReq} \bullet \\
\quad \langle \textit{listReq} \lhd \textit{invoiceInReq}, \textit{listReq} \lhd \textit{customerInReq} \rangle \operatorname{partitions} \textit{listReq}
\end{array}
$$

INIT

$$
\begin{array}{l}
\textit{collectedReq} = \varnothing \\
\textit{plugin4Inv}.\textit{INIT} \\
\textit{plugin4Cust}.\textit{INIT}
\end{array}
$$

respond

[**undefined**]

$$
\begin{array}{|l}
\hline
\textit{collectReq} \\
\Delta(\textit{collectedReq}) \\
\textit{net?} : \textit{clsNetInterface} \\
\hline
\textit{net.reqToReceive} \neq \varnothing \\
(\forall\, \textit{id} : \textit{identifier};\ \textit{req} : \textit{Request} \mid \textit{id} \mapsto \textit{req} \in \textit{net.reqToReceive}) \bullet \\
\quad \textit{collectedReq}' = \textit{collectedReq} \oplus \{\textit{id} \mapsto \textit{req}\} \\
\quad \textit{net.reqToReceive} = \textit{reqToReceive} \setminus \{\textit{id} \mapsto \textit{req}\} \\
\hline
\end{array}
$$

$$
\begin{array}{l}
\textit{IN}1 \mathrel{\widehat=} [\textit{id?} : \textit{Identifier};\ \textit{inv?} : \textit{Invoice}] \bullet \textit{plugin4Inv.s}1 \\
\textit{out}1 \mathrel{\widehat=} \textit{plugin4Inv.out}12 = \text{false} \\
\textit{out}2 \mathrel{\widehat=} \textit{plugin4Inv.out}12 = \text{true} \\
\textit{IN}2 \mathrel{\widehat=} [\textit{id?} : \textit{Identifier};\ \textit{cust?} : \textit{Customer}] \bullet \textit{plugin4Cust.s}2 \\
\textit{out}3 \mathrel{\widehat=} \textit{plugin4Cust.out}345 = 3 \\
\textit{out}4 \mathrel{\widehat=} \textit{plugin4Cust.out}345 = 4 \\
\textit{out}5 \mathrel{\widehat=} \textit{plugin4Cust.out}345 = 5
\end{array}
$$

The class inherits variables from the class *ClsGlobalVariables*. It defines the component *collectedReq* to keep the set of incoming requests that are pending, for processing. The two variables, *plugin4Inv* and *plugin4Cust* define respectively, a reference to an object of the class *ClsCheckInvoice* to check invoices, and *ClsCheckCustomer* to check customers. The set of pending requests is empty initially.

The operation *collectReq*, collects incoming requests from the network, and transfers them into the set *collectedReq*. After checking an invoice or a customer, the function *respond*, forwards the result via the network. Similarly to the class *ClsHelper*, depending on the request under consideration, $IN1$ and $IN2$ each specify a point of connection to activate the sequence of operations, defined in an object of the class *ClsCheckInvoice* or *ClsCheckCustomer* whenever appropriate values are provided by the system environment to the input variables inside the square brackets. The components $out1$, $out2$, $out3$, $out4$, and $out5$ contain the results after a request is processed, and may guide further actions to be taken to effect a scenario.

Next the class *ClsUpdatePoint* is presented .

The class *ClsUpdatePoint*

This class specifies the functionalities to estimate (*evalItem*) the actual value of a returned item, and to update (*updateCustAccount*) a customer's account when, for example, a customer has issued a payment at an agency. The state of the class includes the variable *returnedItem*, to record the set of returned items. An object (*interface*) of the class *ClsLocalSales* (see Section 6.3.3) is included to ensure, for example, that any returned item

was effectively provided by the agency to which it is returned.

ClsUpdatePoint

$\lceil(\mathit{INIT}, \mathit{returnedItems}, \mathit{evalItem}, \mathit{updateCustAccount})$

$\mathit{returnedItems} : \mathit{Item} \times \mathit{Customer} \pfun \mathit{Date}$
$\mathit{interface} : \mathit{ClsLocalSales}$

$\forall\, \mathit{customer} \in \operatorname{ran}\operatorname{dom} \mathit{returnedItems} \bullet$
$\quad \mathit{interface}.\mathit{findCustomer}(\mathit{customer}, \mathit{resp}!) \Rightarrow \mathit{resp}! = \mathrm{true}$

INIT

$\mathit{returnedItems} = \varnothing$
$\mathit{interface}.\mathit{INIT}$

evalItem

$\mathit{item}? : \mathit{Item};\ \mathit{amnt}! : \mathit{Money}$

[to be specified]

updateCustAccount

$\mathit{amnt}? : \mathit{Money};\ \mathit{account}? : \mathit{Account}$

[to be specified]

As mentioned earlier, it may not be necessary to present a complete specification of functionalities, as these may not bring any useful information to evaluate the impact of a UCM model in the construction of a Z and Object-Z specification (see research questions RQ 2 and 3 in Chapter 1).

Next the class *ClsBeneficiary* is presented

The class *ClsBeneficiary*

The class *ClsBeneficiary* derives from the UCM component *Beneficiary* whose role is to monitor the activities modelled by the UCM sub-components *Check_Point* and *Update_Point*. The Object-Z class schemas derived from those sub-components (*ClsCheckPoint* and *ClsUpdatePoint*) were presented earlier.

ClsBeneficiary

$\lceil(\textsc{Init}, receivItem, deliverItem, refundCustomer, e1, e2, e3)$

ClsCheckPoint

ClsMainStartPoints

$deliveredItems : Item \times Item \times Customer \pfun Date$

$update : ClsUpdatePoint$

$\forall item1, item2 : ITEM;\ customer : CUSTOMER \mid$
$\quad (item1, item2, customer) \mapsto date \in deliveredItems \bullet$
$\quad (item2, customer) \in \dom update.returnedItems$

Init

$deliveredItems = \varnothing$

$update.\textsc{Init}$

receivItem

$\Delta(update.returnedItems)$

$item? : Item;\ customer? : Customer;\ date? : Date;\ resp! : Message$

$plugin4Invoice.findCustomer(customer?, sol!) \land sol! = \text{true}$

$update.returnedItems' = update.returnedItems \cup \{(item?, customer?) \mapsto date?\}$

$resp! = ReturnedItemReceived$

deliverItem

$\Delta(deliveredItems)$

$item? : Item;\ customer? : Customer;\ date? : Date$

$item? \in \dom update.returnedItems$

$returnedItems' = returnedItems \cup \{(item?, customer?) \mapsto date?\}$

refundCustomer

$customer? : Customer;\ amount? : Money$

[specify operation to refund a customer]

e1

$\Delta(Scenarios, items, customers)$

$id? : Identifier;\ item? : Item;\ inv? : Invoice;\ cust? : Customer$

$scenarios' = scenarios \ranres \{id? \mapsto sceneReturnItem\}$

$items' = items \ranres \{(item?, inv?, ag?) \mapsto sceneReturnItem\}$

$customers' = customers \ranres \{(cust?, compagny?) \mapsto sceneReturnItem\}$

e2

[specify operation similar to that in e1 for *sceneReplaceItem*]

$e3$

[specify operation similar to that in e1 for *scenePayCredit*]

let $listReq == \text{ran}\ collectedReq$

$[\forall\, id : Identifier;\ inv : Invoice \mid \exists\, req : Request \bullet$
$\quad id \mapsto req \in collectedReq \land req \mapsto inv \in listReq \lhd invoiceInReq] \bullet$
$\quad IN1 \fatsemi (out1 \land e12a) \ [] \ ([acceptOp : Boolean] \bullet$
$\quad\quad (out2 \land acceptOp) \fatsemi respond$
$\quad\quad\quad \fatsemi\ receivItem \land update.evalItem$
$\quad\quad\quad \fatsemi\ (refundCustomer \land e1$
$\quad\quad\quad\quad [] \ deliverItem \land e2)$
$\quad\quad [] \ (out2 \land \neg\ acceptOp? \land e12b)$

[specify a similar sequence of operations for a scenario to pay credit]

This class inherits properties and operations from the class *ClsCheckPoint*, and components from the class *ClsMainStartPoints*. The state schema of the class *ClsBeneficiary*, contains an object of the class *ClsUpdatePoint*, to inherit its operations and to access (by inheritance) the component *returnedItems*, which is updated by the operation *receivItem*. The component *deliveredItems* is defined to specify the set of items sent to customers for the replacement of returned items. The predicate part of the state schema stops the agency from delivering a replacement item to a customer only once the returned item has been collected. Initially, the set of delivered items is empty.

The operation *receivItem* is specified to collect returned items, shipped from distant companies. A new item is added to the list of returned items (*returnedItem*). The operation *deliverItem* specifies the activity that consists of sending an item to a customer to replace the one that the customer returned. It adds the item to the set of delivered items *deliveredItems*. Similarly, the operation *refundCustomer* specifies the activity that refunds a customer for a returned item. The operations $e1$, $e2$, and $e3$ each specify, the reaction of the system when a scenario is terminated successfully. Such operations may be simple or complex, depending on the situation under consideration. For example, it may be the right time to think about archiving all the documents that were involved, or removing or destroying all the documents or information that were temporarily used to assist the operation, and/or any other resource that is of no use after the scenario is terminated. For illustration purposes, $e1$ is specified in detail, whereas, $e2$ and $e3$ are left unspecified, as they can be similarly defined.

Similarly to the class *ClsHelper*, an expression is used to specify the sequence of operations

needed to perform a scenario to return, or to replace an item. A pair of square brackets, enclosing input variables, specifies a place where some actions are needed from the environment, for example, to select some specific values from a given list, or to provide values for the variables. For example, the expression:

$$[\, acceptOp : \mathbb{B} \,]$$

specifies a place where the system requires an action from the environment, e.g., an operator to indicate with a boolean value, whether the operation must continue or not.

6.4 Chapter summary

This chapter aimed to demonstrate the applicability of the framework proposed in the previous chapter, by applying it to the UCM model of the case study (see Figure 6.1), developed in Chapter 3. The purpose was to generate the UCM-OZ version of the Object-Z specification of the case study (see Chapter 1, Figure 1.3).

In line with the recommendations in the framework, the UCM model was sub-divided, by means of stubbing techniques, into three sub-maps, depicted in figures 6.3, 6.4, 6.5 and one principal map (see Figure 6.2). Figure 6.6 was also presented to reveal the connection between the sub-maps and the principal map. During the transformation process, each map was treated individually; starting with sub-maps, followed by the principal map. Since the map in Figure 6.2 was more complex than others (contained more varieties of UCM elements), Figure 6.7 was presented to reveal the hierarchical structuring of its components. The transformation process led to an Object-Z specification, namely UCM-OZ.

The next chapter proposes a generic framework to guide the validation process of a software specification.

Chapter 7

A Framework for validating a software specification

In Chapter 4, an Object-Z transformation of the Z description of the case study was developed. Chapter 6, proposed another version of the Object-Z specification, obtained by transforming the UCM model, of the same case study. This chapter presents a generic framework to evaluate a software specification. This framework is used in subsequent chapters to evaluate the qualities of the Object-Z specifications of the case study, since this dissertation is about evaluating two different paths (seen in Figure 1.3) to address the research questions $RQ2$ and $RQ3$ (Chapter 1, Section 1.2).

The layout of the chapter is: a brief analysis of the conceptual relationship of a specification is first presented with reference to stakeholders (Section 7.1.1), the application domain (Section 7.1.2), language notation with tool support (Section 7.1.3) and the envisioned system (Section 7.1.4). Section 7.2 briefly outlines the difficulty of evaluating a comprehensive set of characteristics for a quality software specification. A "spiral strategy" is proposed in Section 7.3 to guide the validation of a specification. As part of the strategy there follows a brief analysis of the scope of the system. This is followed by an iterative process, consisting of validating the input specification with respect to the expectations of the stakeholders, (Upward validation) in Section 7.3.2, consideration of the application domain (Leftward validation) in Section 7.3.3, the specification language and tool support (Rightward validation) in Section 7.3.4, and the final product (Downward validation) in Section 7.3.5.

In Section 7.4 a two-step mechanism that exploits the result of the validation framework, to compare two specifications of the same set of requirements is proposed. Finally, a brief summary of the chapter is presented in Section 7.5.

The main ideas of this chapter constitute one of the important contributions of this dissertation. The summary was compiled into a full research paper, which was presented at the South African International Conference for Computer Scientists and Information Technologists (SAICSIT'10)(Dongmo and van der Poll [24]).

7.1 Conceptual relationship in a Software specification

Ever since, Brooks [15] published his classic text, in which he stated the importance of conceptual concepts in software development, research in Requirements Engineering has becomes evermore important. As mentioned by Nuseibeh and Easterbrook [65], software ought not to be isolated from the system in which it operates, that is, the application domain. It is commonly accepted that the success of a software system is highly related to the extent to which it meets stakeholders' expectations. Thus, the importance of a proper consideration of stakeholders' needs during a software specification validation process is essential. The emergence of software specification notation languages and associated tool supports, implies the importance of those languages in software specifications. Therefore, to determine the validity of a software specification, we consider analysing, at the requirements level, the conceptual relationships between four aspects. These are stakeholders; the application domain; the specification language and tool support; and the envisioned operational system. Figure 7.1 illustrates these relationships.

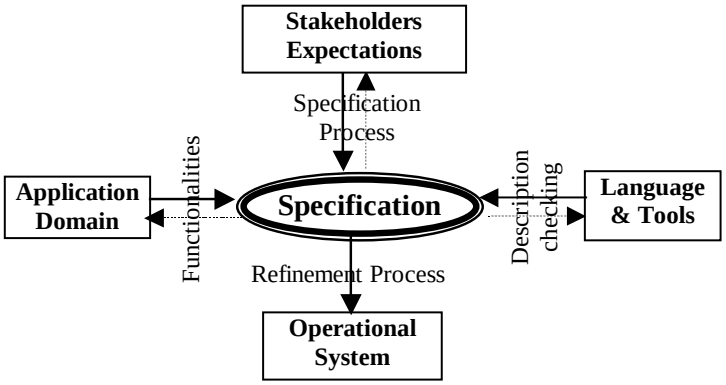


Figure 7.1: Conceptual relationship

Rectangles in Figure 7.1 represent participants in the relationship. The ellipse represents the specification, and arrows pointing to it identify those participants contributing to the construction of the specification document. The arrow pointing to the **Operational System** indicates that the specification itself, participates in the construction of the envisioned software system. The arrows with dashed lines indicate possible feedbacks from the speci-

fication, reporting on, for example, agreements or disagreements with the party which the arrow is pointing at.

The contribution of each of the participants in the relationship is discussed next, starting with the **Stakeholders**.

7.1.1 Stakeholder expectations

Stakeholders denote different people (customers, users, developers, etc.) fulfilling different roles, and often having contradictory requirements which they expect a system to fulfill. For instance, a customer paying for a system expects it to produce benefits within a given time. A user expects the system to be user-friendly and include appropriate functionalities. It is well known that these two categories of stakeholders, often do not know exactly what they require, and fail to express their needs clearly and unambiguously. The development team on the other hand, has the responsibility to produce, within a reasonable time and budget, a system that unambiguously responds to such expectations (see Schach [74]).

During the early phases of a software development process, the specification document plays a vital role among stakeholders. One difficult objective is to construct a specification that satisfies all stakeholders (see Nuseibeh and Easterbrook [65]). Much work in this area has been conducted (e.g. Jureta et al. [45]). Other software verification and validation (V & V) techniques appear in (Dupuy-Chessa and Bousquet [29], McComb and Smith [56], Plagge and Leuschel [69], Sargent [73], Schaefer and Poetzsch-Heffter [75]). Popular trends in requirements elicitation favour the active participation of all stakeholders in the process. Notable examples are JAD[1](see Wood and Silver [99]) and placing the developer in the working environment of a stakeholder, as discussed by Friedrich and van der Poll [31].

Since a specification may result from successive refinements of stakeholder expectations (Van Lamsweerde [96]), it is argued that the adequacy of such a refinement process ought to be subject to a validation. Hence, any attempt to validate a specification against initial goals should consider both the specification, and the goal refinement processes. Such an approach may facilitate the identification of, not only the expected characteristics of an appropriate specification, but also those of a reliable goal refinement process.

[1]Join Application Development

7.1.2 The Application domain

An application domain is here considered as the operational software environment, and is, therefore, the source of various types of information (processes, services, data, actors, etc.). Naturally, these are needed to define the boundaries and functionalities of the specification. Work in the areas of domain analysis and modelling (e.g. Evans [30], Miller [61], Valerio et al. [88]), attempts to make such information readily available and reusable. Techniques such as the Service Oriented Architecture (SOA by Brown [16]) attempt to modularise domain information into services. Hence, given a set of objectives to be achieved by an envisioned software product, an appropriate specification technique would consider all the appropriate information in the domain, to construct the specification. It is argued that a good software specification should therefore, be generated from domain-traceable information, and include functionalities that may be integrated into well-defined operational services of the problem domain (e.g. principles 3 & 4 in Balzer and Goldman [10]). For example, if a specification includes an operation, such an operation should be clearly part of a complete scenario, or use case, in the system domain.

On the strength of the above arguments it is proposed that the validation process of a specification ought to:

- Ensure that each described operation is traceable from the application domain and feeds into an operational service.

  This may avoid inappropriate reuse of specification fragments. E.g., a careless reuse of the specified operation "Buy parts" that satisfy the goal: *The system shall benefit the usage of spare parts* in a car garage, may have adverse effects in a domain such as medical equipment repairs, where safety-critical measures should be considered, despite the fact that the same goal is to be achieved.

- Ensure that the context and the purpose of each operation is unambiguous.

  This may facilitate the usability and appropriateness of each operation. Preconditions have to be determined to reveal when and where (e.g. within what service, scenario, or use case) the operation to be used is, and that it does not conflict with other, similar operations.

7.1.3 Languages and tool support

Specification languages and their tool support are, important building components in the ultimate system (Nuseibeh and Easterbrook [65]). Natural languages, being inherently am-

biguous (see e.g. Kamsties et al. [46]), can only play a supportive role in software specification and modelling. Since specification languages are generally more cryptic than natural languages, some of the key properties expected from a good specification (e.g. readability, understandability, etc.) are highly related to the encoding capability of the language. For example, as indicated in Chapter 2, the concept of class schema added to standard Z, yielded Object-Z (Duke and Rose [26], Smith [77]), thereby adding object orientation to Z.

Specification languages generally differ from each other, according to their encoding capabilities, supportive techniques, methodology or associated tools and may, therefore, not be equally applicable in all situations. For instance, as mentioned in Chapter 1, at a high level of classification, mathematical-based languages may bring more details into specifications, and eliminate errors and ambiguities in requirements, but need more effort to be comprehended and hence, are, arguably, more suitable for safety-critical problems. On the other hand, semi-formal languages are more user-friendly, and relatively easier to manipulate by humans, but do not have the ability to completely eliminate or detect defects and ambiguities in the statement of requirements. More importantly, some specification notations are appropriate to describe functional properties of a software system, e.g. UCMs, Z, Object-Z, etc., whereas others are good at describing non-functional properties (e.g. Goal-Oriented Requirement Language (GRL[1])).

More than one language may, therefore, be needed for a comprehensive specification of software systems (Amyot and Mussbacher [7]). Nowadays, much of the work on the validation of specifications constructs, focus, *firstly*, on the validation of functionalities against the initial user-requirements and, *secondly*, the validation of language-related properties (e.g. internal consistency: type-checking, semantics, etc.). Despite the fact that the scope of this work does not deal with validating specific language-related problems, the second step of validation depends on the available tool support for the language. It is plausible that the use of such tools would contribute to the quality of the final specification. Considering the above analysis, the following to support the validation process of a specification is suggested:

- Ensure that, at the requirements phase, the language and tool support are appropriate and applicable to the problem domain.

  The notation used, ought to include all the capabilities required to adequately describe information taken from the problem domain, in a way that satisfies stakeholder expectations.

- Ensure the integrity of the whole set of components, including the language, tools, and the associated methodology.

If the integrity of any part of the process is problematic, the quality of the final specification document(s) may be affected. For example, consider a situation where a set of initial requirements are first transformed into Use Case Map diagrams (UCMs), then converted to UML (Booch et al. [11], Selic [76]) diagrams, and from UML to standard Z. The transition between UCMs and UML diagrams may not be harmful, because both notations support Object-Orientation. But, the transformation from UML to standard Z may be problematic owing to the difficulties of matching UML Object-Oriented concepts with standard Z, which does not explicitly support Object-Orientation; this is unless an appropriate transformation framework from UML to Z is provided, as the one presented in Chapter 5 for UCM, Z and Object-Z.

- Ensure that the tools used to support the specification (e.g. type-checkers, theorem provers, semantic analysis, etc.) are suitable for the task. For example, a specification language may be too "rich" to directly serve as input to a theorem prover, as indicated by van der Poll [90].

7.1.4 The envisioned final product

If a software specification is considered as a set of defined properties, and a software product as a set of implemented properties, an ideal for software verification would be to create a bijection between the two sets. Since a verification exercise is concerned with demonstrating that the implementation is correct with respect to requirements (Heimdahl [36]), this would consist of partitioning the user requirements, to form a set, and partitioning the implemented components to form another set in such a way that each element in one set, is bijectively related to an element in the other set. This would indicate that no extra functionality has been introduced into the implementation, and no requirement was omitted during the implementation. Such a technique may be possible at the verification phase, since both the requirements and the implementation thereof are readily available. In contrast, at the validation phase, which is the subject of this work, only the specification is available. This creates a challenge, since one would be trying to demonstrate, at least, that there exists a surjection from the set of implemented properties (not yet available) to the specified properties at hand.

Instead it is suggested to focus on the validation of a specification to establish the existence of appropriate refinement or design/implementation techniques, and the suitability of the specification, to serve as inputs to such techniques. This may consist, for example, of identifying, at least one applicable refinement technique, and proving that it couples well with the technique(s) and method(s) on which the specification is based. For instance, depending on the situation at hand, such a process may refer to a direct refinement technique

(automated or not), or to traditional design and implementation processes of the software development life cycle. To illustrate the idea, consider coupling, in sequence, UML and Turbo Pascal to implement a Z specification, where UML is used to design the system, and Pascal is used for coding. It may be argued that the ability of a Z document, to map onto UML, is suspect because Z is not inherently object oriented, unless an appropriate framework (e.g. similar to the one proposed in Chapter 5), is provided to perform the transition from Z to UML. For similar reasons, the process of implementing UML diagrams with Pascal may also be problematic.

The question of identifying the characteristics of a good specification is addressed in the next section.

7.2 Characteristics of a quality software specification

Since software systems generally have different goals and expected behaviours, it is a complex task to compile a complete list of the properties of a good specification. Furthermore, each specification targets a particular aspect of the problem being specified (e.g. domain description, user view of the problem, problem solution, etc.) and has a precise purpose (e.g. communication, documentation, etc.). Therefore, a specification is only as good as the degree to which it achieves its intended goals and fulfills its purpose. However, some minimum standard may be expected from any satisfactory specification, (e.g. completeness, consistency and correctness exemplified by IEEE Std 830-1998 [41]). Since the specification itself results from the needs analysis process, it may be argued that the quality of a specification in turn depends on the quality of the elicitation process. In line with this reasoning, some desirable characteristics were proposed for requirements elicitation procedures by van der Merwe and Kotzé [89]. Although the focus of this work is not explicitly on requirements elicitation techniques, it is suggested that in case of defects in the specification, the quality of the specification process should be measured against similar characteristics.

It may be observed that the four aspects of a software system specification represented in Figure 7.1, are inherently inter-dependent. For example, to achieve a stakeholder's expectation (goal), some services or operations from the application domain, together with the constraints on their applicability (policies, standards, etc.), need to appropriately encoded with a notation language. This implies that a validation process that may consider each of the four aspects at a time is iterative, in nature since a change on one aspect of the specification, may affect others. In this regard, an iterative validation process, based on Boehm's spiral model is proposed next.

7.3 Validating a specification document

Figure 7.2 summarises the proposed basic spiral validation mechanism. The spiral indicates

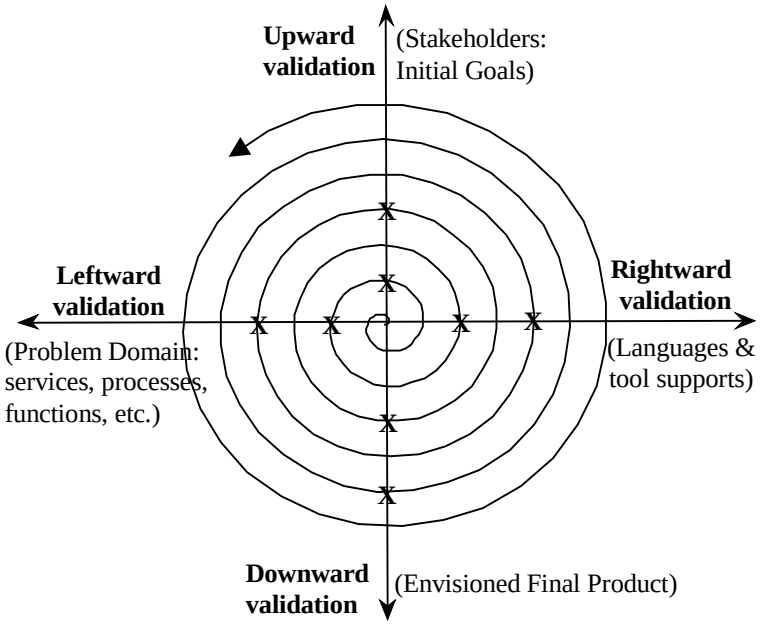


Figure 7.2: Basic validation strategy

the iterative progression of activities along those axes. The first iteration of the spiral starts from a point close to the stakeholders segment and the intersection point between the four quadrants, but stops at the language axis, which is not explicitly part of the validation process. It is included simply to indicate, for example, the fact that some work has been carried out involving the four aspects to generate the specification. The validation process is, therefore, seen as the continuation of such work.

Four types of validations are considered along four axis. A cross (X) indicates a validation point where one validation phase ends and the specification amended if necessary. The arrow at the end of the spiral indicates a possible order in which activities are carried out. The iteration is motivated by the fact that the results at one checkpoint, may change the original inputs (specification, tools, methodologies, etc), and hence, induce the repetition of other activities. The upward validation considers the properties of the specification relative to stakeholders' expectations. The leftward validation is concerned with the conformity of the specification to the problem domain. The rightward validation concentrates on the language and tool-related properties. The downward validation considers the appropriateness of the specification in relation to its intended use.

Next the validation scheme is discussed in further detail.

7.3.1 The scope of a specification

Initially the scope of the validation aiming to clarify aspects of a specification to be evaluated, is defined. For example, if the main purpose of a specification is to facilitate communication between stakeholders, the validation process may concentrate on those properties that contribute to communication, hence the need to perform a preliminary check on the scope and initial objectives. Any redundant functionalities in the specification identified during the preliminary check, should be removed. The case of the specification not sufficiently describing the initial requirements is addressed by The **upward** and **leftward** validations phases as those two phases are concerned with evaluating the specification against stakeholder expectations, and the application domain. To summarise this phase, the following is suggested:

- Define the scope of the validation to be carried out.
- Eliminate, wherever possible, redundant functionalities in the specification.

Next the upward validation phase is described .

7.3.2 The upward validation

The objective of this validation phase, is to ensure that the specification includes all the necessary information to satisfy stakeholders' expectations. Depending on individual cases and the main purpose of the specification, it is proposed to (clearly) list and describe all the properties required from both the specification, and the specification processes used, before continuing with their validation. For example, the following properties may be expected:

- For the specification: Completeness and Adequacy.
- From the specification process: Traceability and Reliability.

Referring to work of van Lamsweerde [95], a specification is *complete* with respect to the initial goals, if the collection of properties in the specification is sufficient to establish such goals. In turn, the *adequacy* of a specification indicates the extent to which the specification clearly addresses the problem at hand.

Although much of the non-functional requirements stem, in general, from the application domain (domain constraints) and the expected behaviour of the final product, this phase is the appropriate juncture to get them validated. Issues concerning, for instance, the user-interface, usability, or the performance of the system, can be investigated with the participation of potential users and other stakehoders. At this stage, new notation language or

tools may be considered, to adequately describe such functionalities that were omitted in the initial specification.

In some cases, the specification techniques are determined by available language notation. Here, the validation of the properties (of the specification) related to the specification process may, be deferred to the rightward validation phase. Such a validation may also involve more than one technique or notation, since the initial requirements may need to be transformed into system requirements, before being specified (see Figure 1.2 in Chapter 1). In the latter case, the integrity of the techniques needs to be established.

7.3.3 The leftward validation

This phase demonstrates and measures the applicability of the specification. On the strength of the analysis in Section 7.1.2, the following are suggested:

- Ensure the usability of each specified object (dynamic or static). For each object, or set of collaborating objects, ensure that there is at least one service, process, activity, use case, or scenario in the problem, domain that needs it.
- Ensure that high-level domain constraints, related to the use of each object specified, are well described (e.g. adherence to standards and policies).
- Ensure that the context and purpose of each object specified is clear and unambiguous, relative to its usage.
- Ensure that executable objects do not conflict each other during their execution.

Next the rightward validation phase is presented.

7.3.4 The rightward validation

This phase is generally the focus of most of the verification and validation techniques. The emergence of different types of specification notations has led researchers to continuously invest effort to improve the quality of specifications, and hence, the creation of valuable analysis tools. The purpose is to ensure that a specification is both syntactically and semantically correct, well structured, and has the ability to serve as input to the analysis tools. The following is proposed:

- List all the system properties that are required from the specification, and clearly and succinctly describe each of them to make their purpose unambiguous.

- For each property, identify an appropriate method or technique as well as supporting tools.
- Proceed with the validation, aimed at establishing the extent to which each property listed, is available in the specification.

As an example, consider the validation of a specification, whose primary role is to document a user-view of a software system, and facilitate the discussion between stakeholders. We may expect the specification to include the following properties: correctness, internal consistency, readability, understandability, communicability and minimality (i.e. no redundant constructs). These properties are considered briefly below:
Correctness: Check that the specification is syntactically and semantically correct. This calls for the use of a syntax checker and analysis tools for the semantics.
Internal consistency: Identify and discharge necessary proof obligations.
Readability: Inspect the specification for issues of structure. Such an inspection might not need tools.
Understandability: As for readability.
Communicability: Related to readability and understandability.
Minimal: Check whether the specification contains only essential objects.

The following section describes the downward validation phase.

7.3.5 The downward validation

This process considers to what extent the specification will satisfy the desired quality of the final product. In the light of the analysis in Section 7.1.4, it is arguable that the core of this work relies on 3 major components: the specification itself, the transformation techniques/methods to shift from one stage to another (see, for example, the framework in Chapter 5), and the final product. The following is suggested:

- List all the system properties that are required from the specification, and clearly and succinctly describe each of them.
- Identify for each property, an appropriate method or technique with, when necessary, tool support.
- Identify applicable transformation (or refinement) techniques and tool support.

  N.B. in practice such techniques and tools might already have been determined during an earlier phase, e.g. planning. Still then, the argument is that the specification validation phase is the correct juncture to proceed with a systematic feasibility study.

- List key properties that are expected from those techniques and tools.
- Proceed with the validation process.

For example, if the purpose of a specification is to build a traditional software system, the following may be relevant:

1. The specification is to be: executable, minimal, well formed, etc.
2. Validation techniques: Inspections, Round table discussions, Animation (see McComb and Smith [56]), Prototyping, Automated proofs.
3. Validation tool: A suitable type-checker.
4. Some appropriate design / implementation, or refinement techniques available.
5. Tools to illustrate some HCI[2] properties: e.g. usability.

Next is briefly presented how this framework may be exploited to compare specifications.

7.4 Comparing specifications

The above framework suggests the following steps are necessary to compare specifications:

- Apply the validation steps described in Section 7.3, to each specification, and document the result of the validation.
- Proceed with the analysis of the validation results, and provide feedback based on the analysis.

The following section concludes this chapter.

7.5 Chapter summary

This chapter presented a framework to develop and validate a software specification. Firstly, the conceptual relationship of a specification with reference to stakeholders, the application domain, language notation with tool support and the envisioned final product is described. Then, a generic 4-way validation strategy that uses Bohem's spiral model to develop and measure a software specification against requirements from users (upward validation), the

[2]Human Computer Interaction

application domain (leftward validation), restrictions in language notation and tool support (rightward validation) and finally the operational system (downward validation) is proposed.

The summary of the proposed framework was compiled into a full research paper, which was published in the proceedings of the SAICSIT'10 Conference(Dongmo and van der Poll [24]). This chapter therefore, constitutes one of the major contributions of this work.

The following chapter, applies the framework here developed to validate each of the two Object-Z specifications of the case study (described in Chapter 3): Z-OZ and UCM-OZ developed, respectively, in Chapter 4, and Chapter 6, following two alternative specification processes (see Figure 1.3 in Chapter 1).

Chapter 8

Applying the framework to the Case Study

This chapter applies the framework proposed in the previous chapter, to validate the two Object-Z specifications Z-OZ and UCM-OZ developed, respectively, in chapters 4 and 6. The validation scheme developed in Section 7.3 is initially applied to the Z-OZ specification and then, to the UCM-OZ specification. During the validation process, only one iteration is performed in each case.

Section 8.1 reviews the stakeholder expectations and user requirements for the case study. The scope of the validation is presented in Section 8.2, and a common list of properties and criteria to judge the quality of each of the two specifications are defined, respectively, in sections 8.2.1 and 8.2.2. Section 8.3 presents the validation of Z-OZ specification and Section 8.4 the validation of UCM-OZ. Finally, Section 8.5 summarises the chapter.

8.1 Expectations and user requirements

This section reproduces the list of expectations from stakeholders and user requirements described earlier, in Chapter 3, in natural language (English). It is important to explicitly identify such lists, since the quality of each of the specifications under consideration depends on them.

8.1.1 Expectations of stakeholders

According to the case study description (see Chapter 3), stakeholders expect the following from a successful specification for the system:

1. A specification that facilitates their understanding of the system.

2. A specification that stimulates a thorough discussion among stakeholders about the feasibility of their proposed idea to help each other to gain mutually.

3. A specification that exhibits a possible way to render the above idea of collaboration operational.

The above expectations suggest some qualities that a specification ought to encompass to be satisfactory. For example, a specification that includes properties pertaining to communication, such as Understandability, Readability and Clarity, may go some way to satisfy the primary expectations of stakeholders i.e. facilitate their understanding of the system. Such properties are identified and presented in subsequent sections.

The next section discusses the user requirements presented in the case study.

8.1.2 User requirements

This section presents the three main user requirements of the system, as discussed in the case study in Chapter 3. Those requirements are the following:

(a) Return of an item:
To allow a customer to return a purchased item at any company. When the item is received, some operations, involving the local company and the provider of the item, must be performed internally to complete the returning of the item.

(b) Replacement of the item:
Similarly to returning items, this service allows a customer to replace a purchased item at any agency in the customer's local area. When the item is received, some operations are performed internally involving the local agency, and the provider of the item aiming to complete the replacement process.

(c) Pay credit:
To allow a customer to pay a credit at any agency in the local area. When such a payment is made at an agency, some operations are carried out internally to complete the credit payment.

Note that internal activities between agencies are not defined in the case study. Initially those are not known by the stakeholders and one of the responsibilities of the system specifier is to propose them. The ability of a specification technique or method to help define such functionalities, or stimulate thinking about them is, therefore, an important aspect that ought to be considered during the validation process.

8.2 The scope of the validation

As mentioned in the introduction to this chapter, this validation process uses the framework proposed in Chapter 7, as a guide, to investigate the qualities of an input specification. The focus is to measure the extent to which the specification satisfies stakeholders' expectations (see Section 8.1.1), and to evaluate the appropriateness of the specified operations with regard to achieving the user requirements, listed in Section 8.1.2. To these aims, a sample list of properties to be investigated is considered. These include: Readability, Understandability, Clarity, Traceability, Completeness, Operational feasibility, Creativity, Networking, Integrity, Correctness (with regard to the encoding of the specification), Internal Consistency, Minimality, Maintainability, Technical feasibility, and Executability.

In the next section, those properties are grouped according to the validation phase (see Chapter 7) in which they are discussed. Some clarifications are also presented about the context of each of them, before the proper validation process is later described.

8.2.1 Properties to be validated

- The *Properties related to Stakeholders' expectations* are: The readability, understandability, clarity, traceability, completeness and the operational feasibility. These properties are considered during the **upward validation** phase (see Chapter 7, Figure 7.2 or Section 7.3.2).

  The **readability** pertains to the ease with which the specification can be read, whereas the **understandability** refers to the extent to which the specification can be understood. The **clarity** refers to the expressiveness and structuring of the specification. Together, these three properties help to evaluate the ability of the specification to be communicated, and address the first two expectations (section 8.1.1). The **traceability** is related to the specification process. It measures the ability of the process to preserve information, aims to detect requirements omitted, or those that were changed during the specification process. **Completeness**, refers to the need to ensure that the specification is sufficient to establish the initial (user) requirements. The **operational feasibility** ensures the specification can be effected in practice, and facilitate estimates and managerial tasks such as the planning of activities and work schedules. When encompassed by a specification, the operational feasibility would contribute to the achievement of the stakeholders' expectation number 3, above (see Section 8.1.).

- The *properties related to the application domain* are: Creativity, networking and in-

tegrity. These properties are validated during the **leftward validation** phase of the framework proposed in Chapter 7.

Creativity implies the idea of innovation, in the sense of identifying or proposing appropriate functionalities to accomplish the user needs. This quality indicates whether the specification at hand includes such activities needed to accomplish each of the user level services listed in Section 8.1.2. This property contributes to measure the extent to which the specification includes sufficient detail functionalities to describe the required services. Additionally, it also helps to achieve the third expectation (Section 8.1.1), as it requires the specification to suggest a way to build an operational system, from the initial idea concepts. The **networking** property measures the ability of the proposed specification to handle network issues related to distant communication between dispersed agencies. The idea of network communication is suggested by the structure of the agencies, as presented in Figure 3.1(Chapter 3). The issue with the **integrity**, is to investigate the applicability of the functionalities described in the specification. The presence of these last two properties in a specification may also provide insight about the extent to which user level services can be achieved, as they give an indication of the network services handled, and the integrity of the dtail functionalities of the system.

- The *Properties related to the notation language and tool supports* are: correctness, internal consistency, minimality and maintainability. These properties are considered during the **rightward validation** of the validation framework proposed in Chapter 7.

  As mentioned in Section 7.3.4, (Chapter 7), the **correctness** property checks that the specification is syntactically and semantically correct. The **internal consistency** indicates that the specification has a meaningful semantic interpretation that makes true all specified properties taken together, whereas the **minimality** verifies that a specification does not include properties that are irrelevant to achieve stakeholders' expectations, or do not contribute to define or describe any of the user requirements listed in Section 8.1.2. Finally, for this phase, the **maintainability** refers to the ability of a specification to accommodate later changes.

- The *Properties related to the envisioned product* are: technical feasibility and executability. These two properties are discussed during the **downward validation** of the proposed framework (see Section 7.3.5, Chapter 7).

  The emphasis here is to demonstrate that the final product (i.e. the software system) with the required qualities, can be generated from the specification at hand.

Since the properties identified in the earlier phases aim to establish that the specification includes the required qualities, during this validation phase, the focus here may be on the technical feasibility and the executability of the specification.

Naturally, the list of properties presented above is not exhaustive. However, we may need to establish that such a list is sufficient to demonstrate the validity of a specification relative to the given requirements and stakeholders'expectations. For illustration purposes, assume, for example, that P is the set of properties that a successful specification has. Let G be the set of goals or expectations, the task being to ensure that if a specification includes all the properties in P, then it satisfies G. For example, in the case of the Z-OZ and UCM-OZ specifications, the following may be considered:

- G would include both the list of stakeholders'expectations (see Section 8.1.1) and the user requirements, listed in Section 8.1.2.
- P would include the properties listed above in this section.

To generate the list of properties in P that aim to characterise a specification that satisfies the set of expectations in G, one may proceed with the analysis of the stakeholders'expectations and uses'r requirements.

For example, it may be argued that a documented idea is communicable if the document itself is readable, exhibits some clarity and is understandable; the three properties of Readability, Clarity, and Understandability would be included in P, if a specification is expected to be communicable.

The list of properties presented earlier in this section, and the list of stakeholders' expectations (Section 8.1.1), and users' requirements (8.1.2) are presented in Table 8.1. Codes $p1$, $p2$, ...,$p16$ are associated with properties, whereas, codes $g1$, $g2$, ...,$g6$ are used for goals. The first two columns of the Table 8.1 are used to list the properties, and the next two for the system goals and users' requirements. A group of properties that support the same goals, are in the column on the left, and the supported goals are in the column on the right side.

code	P (Properties)	code	G(Goals and expectations)
$p1$	Readability	$g1$	facilitates the understanding of the system
$p2$	Understandability	$g2$	stimulates a thorough discussion between stakeholders
$p3$	Clarity		about the feasibility of the idea.
$p4$	Operational feasibility	$g2$	stimulates a thorough discussion between stakeholders
$p5$	Technical feasibility		about the feasibility of the idea.
$p6$	Executability	$g3$	exhibits a possible way to render their idea operational.

$p7$	Creativity	$g3$	exhibits a possible way to render their idea operational.
$p8$	Traceability	$g4$	Return item
$p9$	Completeness	$g5$	Replace item
$p10$	Networking	$g6$	Pay credit
$p11$	Integrity		
$p12$	Correctness		(target the system)
$p13$	Internal consistency		(target the system)
$p14$	Minimality		(target the system)
$p15$	Maintainability		(target the system)
$p16$	Modifiable		(target the system)

Table 8.1: Mapping properties to requirements

Table 8.1 shows a mapping between the properties in P, and the list of user requirements and system goals in G. Each mapping relating one or more properties to one or more goals, illustrates that the properties when taken together, contribute to the achievement of the related goal(s). For example, as mentioned above, if a specification is readable, understandable, and clear, then it will therefore require less efforts to be communicated amongst stakeholders. Such a specification would therefore achieve the requirement $g1$ (see Table 8.1 and Section 8.1.1).

If, for example, we consider a property in P, to be either present or absent in a specification, and a goal in G to be either satisfied or not, then the expression 8.1 further describes the logic behind the mapping of properties $p1$, $p2$ and $p3$ to the goal $g1$. The semantics of this formula, is that if the properties $p1$, $p2$ and $p3$ are all simultaneously present in a specification, it is more likely that the goal $g1$ will be satisfied by the specification.

$$p1 + p2 + p3 \rightsquigarrow g1 \tag{8.1}$$

Note that the meaning of the symbol + in Formula (8.1), is not the same as in Formal Logic or arithmetics. It simply indicates that the more properties a specification encompasses, the more the goal(s) on the right side (of the symbol $\rightsquigarrow$) is supported. Note also the use of $\rightsquigarrow$ in this formula to denote *support* for the goal, rather than an absolute claim about the truth of $g1$ using a strict implication or turnstile ($\rightarrow$), the later two symbols traditionally indicate a strong relationship.

As indicated in Table 8.1, the properties involved in Formula 8.1 also partly contribute to the achievement of the goal $g2$, which may be sufficiently satisfied if the specification is

likely to facilitate both the operational and technical feasibility ($p4$ and $p5$) of the system and is executable ($p6$). Thus, the following formula may be generated:

$$(p1 + p2 + p3) + (p4 + p5 + p6) \rightsquigarrow g2 \tag{8.2}$$

In the same vein, the properties $p4$, $p5$ and $p6$ in a specification partly demonstrate the ability of the specification to render an operational software product. However, this ability may be sufficiently inferred if a reasonable refinement of high-level goals and user requirements into system functionalities is proposed ($p7$: creativity). Those are the functionalities that are well integrated ($p11$: integrity) into the application domain, to perform high level services. Such operations or functionalities ought to address issues related to network communication ($p10$), and be traceable ($p8$) and complete ($p9$) to integrate to accomplish each of the three major services of the system. Thus, the following formula is used:

$$p7 + p8 + p9 + p10 + p11 \rightsquigarrow g4 \wedge g5 \wedge g6 \tag{8.3}$$

Formula 8.3 indicates that $g4, g5$ *and* $g6$ are more likely to be achieved by a specification that encompass the characteristics $p7, p8, p9, p10,$ *and* $p11$. The next formula shows those properties that are required for goal $g3$ to be supported.

$$(p4 + p5 + p6) + (p7 + p8 + p9) \rightsquigarrow g3 \tag{8.4}$$

The last five properties in Table 8.1, $p12, p13, p14, p15$ and $p16$ apply to the system as a whole. They imply some qualities that a specification ougt to encompass. The list of such properties can be extended to include those pertaining to satisfy, for example, *non-functional* requirements, such as reliability, performance, usability, security, etc. Since specific (or measurable) goals are not defined for our case study, the following formula may be used.

$$p12 + p13 + p14 + p15 + p16 \tag{8.5}$$

Now consider the following system formed by the formulae from 8.1 to 8.5:

$$\begin{aligned}
&\{\\
&\quad p1 + p2 + p3 \rightsquigarrow g1,\\
&\quad (p1 + p2 + p3) + (p4 + p5 + p6) \rightsquigarrow g2,\\
&\quad p7 + p8 + p9 + p10 + p11 \rightsquigarrow g4 \wedge g5 \wedge g6,\\
&\quad (p4 + p5 + p6) + (p7 + p8 + p9) \rightsquigarrow g3,\\
&\quad p12 + p13 + p14 + p15 + p16\\
&\}
\end{aligned}$$

During the validation phase, the above system of formulae may help to relate specification properties to goals, and therefore, provide guidance to observe the extent to which a specific specification satisfies stakeholders expectations.
Some possible advantages of constructing such a set of formulae are the following:

- for the purposes of automation, it may be used as a guiding protocol to control and monitor the validation process of a proposed specification. For example, after a validation phase, resulting values for the properties being checked, are input and a possible list of non-achieved goals or requirements may therefore be generated.
- the status of the specification, and the state of the validation process may be determined by examining it.
- the set is flexible, and may hence be updated during the validation to accommodate changes.

The following section presents some of the criteria that are used to guide the validation of both the Z-OZ (see Chapter 4) and UCM-OZ (see Chapter 6) specifications of the case study, (refer to Chapter 3 for the case study description).

8.2.2 Validation criteria

The criteria that are considered next were inspired from the idea relating the way humans think, to concepts in Object-Orientation, as observed by Hatton [35].

Based on a mathematical model of human reasoning and some empirical work that aims to compare defects in programs, resulting from procedural languages and the Object-Oriented programming language C++, Hatton [35] concluds that some concepts in Object-Orientation, do not conform to the way humans think.

The model shows how information flows between the short-term and the long-term memory. Any incoming information is temporarily kept in the short-term memory. When successfully encoded, the information is transferred to the long-term memory in which it is permanently stored. So, because the size of the short-term memory is limited, Hatton points out that "encapsulation" only partly matches human reasoning, and that neither inheritance nor polymorphism does so. Although such a conclusion is debatable, it is worth observing that multiple hierarchical levels of inheritance and polymorphism, as well as the (large) size of objects (that may be due to encapsulation), in a software specification, may compromise the quality of the specification, especially when it comes to the readability, the understandability and the clarity. This is because multiple exchanges of information between the short-term and long-term memory is required, for example to identify inherited classes, before the correct information (on the inherited class) becomes accessible to the reader.

In the case of encapsulation, for example, objects or components need to be of a reasonable

size to fit into the short-term memory. Thus, the following criteria are suggested to guide the validation of some of the properties mentioned above:

- Depth or level of hierarchy for inheritance
- Depth or level of hierarchy for polymorphism
- Size of Objects or components

The property $p4$—the operational feasibility in Table 8.1, measures the ability of a specification to estimate the efforts needed for the activities in further development phases. It may be argued that, this property relies, amongst others, on two major aspects of a specification which include the nature of objects within the specification, and the structuring of the specification itself. For example, if the components of a specification are made simple and well structured to clearly reveal interaction amongst them, then the feasibility study of such a specification may be facilitated. In line with this idea, van der Poll and Kotzé [93] proposed in principle #5 to refine each operation in a Z specification into a sequence of "primitives" to maintain high "cohesion" in which, a primitive manipulates at most, one state component. Conversely, a low cohesion indicates the grouping of unrelated activities.

Inspired by the above observation relative to the nature of components in a specification and their structuring, the following may further serve as guiding criteria when judging the quality of a software specification:

- The complexity of each object specified
- The grouping or structuring of objects in the specification

Next is presented the validation of the Object-Z specification derived from the Z description of the case study (see Chapter 4 for the Object-Z specification, Chapter 3 for the case study description and its Z counterpart).

8.3 Validating the Z-OZ specification

In this section, the generic 4-way framework developed in Chapter 7, is used to guide the validation of the Z-OZ specification. During the validation process, each of the four validation phases is applied to the input specification, aiming to exhibit the presence or absence of properties pertaining the measurements of the quality of the specification. As the framework recommends considering both the specification and the underlying process, we begin by briefly presenting them in the next section.

8.3.1 The Z-OZ specification

Figure 8.1 illustrates the two-step specification process of the Z-OZ, which is an Object-Z specification of the case study, for which a detailed description was presented in Chapter 3.

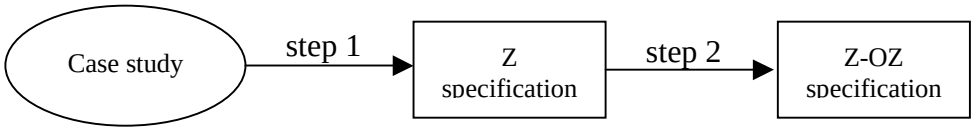


Figure 8.1: Z-OZ specification process

The case study description included the requirements of the software given at the user level, in natural language (English). The first step of the specification, also part of Chapter 3, consisted of translating the initial requirements into a Z specification. At this stage the transformation process was based on the Z Enhanced Strategy (Lightfoot [52], Potter et al. [70]) as well as guidelines provided in the literature to aid the construction of Z documents (van der Poll and Kotzé [93]). The second phase of the specification process, was presented in Chapter 4, and consisted of transforming the Z specification into Object-Z. This process was based on the work of Carrington and Smith [22], Periyasamy and Mathew [68] and the semantics of Object-Z constructs given by Duke and Rose [26] (See Section 4.1, Chapter 4). The Z-OZ specification itself is fully presented in Chapter 4.

The validation of the Z-OZ specification, as well as that of the UCM-OZ document are presented in the following sections. It considers only one iteration (spiral) of the proposed validation model (see Chapter 7, Figure 7.2) aiming to measure their qualities, to enable the comparison without amending the specifications.

8.3.2 The upward validation

This validation phase is concerned to measure the quality of the specification with respect to the properties pertaining to the communicability and operational feasibility. The assumption is that if one can successfully communicate the specification to stakeholders, then, such a specification would, consequently facilitate their understanding of the system; and may therefore be used as a tool to further discuss the system (see formulas 8.1 and 8.2). A further assumption is that a successful specification for the proposed user requirements may provide inputs for the feasibility studies of the envisioned system. In the light of the above assumptions, the validation process, developed next for this phase, targets the following properties: the readability ($p1$), the understandability ($p2$), the clarity ($p3$) the operational feasibility ($p4$), the traceability ($p8$) and the completeness ($p9$) of the specification. The

validation of those properties, starting from the next paragraph, is conducted in the light of the guiding criteria (Section 8.2.2).

To facilitate the analysis of the hierarchical structuring of the Z-OZ class schemas aiming to achieve inheritance and polymorphism, Table 8.2 is used; the table is discussed below.

No.	Class	**Inheritance**				**Polymorphism**			
		1	2	3	4	1	2	3	4
0	*Basic types*	0	0	0	0	0	0	0	0
1	*ClsGlobalVariables*	1	0	0	0	0	0	0	0
2	*ClsAccount*	1	0	0	0	0	0	0	0
3	*ClsIsales*	1	2	0	0	0	0	0	0
4	*ClsDatabase*	1	1	0	0	0	0	0	0
5	*ClsAgency*	1	2	2	0	0	0	0	0
6	*ClsSystem*	0	0	0	1	0	0	0	0

Table 8.2: Levels of inheritance and Polymorphism for Z-OZ

The specification comprises a total of six classes as shown in Table 8.2. This excludes the class of basic types. A class is at a level of hierarchy, say *i*, if it inherits properties or methods from at least one class which is at level *i-1*. In the case of the Z-OZ specification, the highest level for the inheritance is level four (see Table 8.2).

As stated above, Table 8.2 indicates the levels of hierarchy for the inheritance and polymorphism in the specification. Except for the class of basic types, each individual class in the specification is represented. The class of basic types (see Chapter 4) does not inherit from any other class. They are assigned the value zero at each level. Those classes may impede the comprehension a specification, because the only information they convey is their name.

Note that the zeros in the cells for the polymorphism indicate that the concept of polymorphism, is absent from the specification. This may be due to the fact that the specification originated from Z, which does not encompass Object-Orientation in the first place. It may equally be an indication of an insufficient analysis and inadequate description of the relationships between objects or components of the system.

The idea of using hierarchical levels for the inheritance and polymorphism is valuable when

we observe, for instance, that to master an inherited class, it is necessary to keep in mind (short term and long term memories (Hatton [35])), a picture of both the inheriting and the inherited classes. It may become very complex to comprehend a specification that comprises classes with multiple levels of hierarchy in terms of inheritance and polymorphism. In this regard, the class *ClsSystem* (see Table 8.2) appears to be the most difficult to master, as it is at the highest level. To illustrate the complexity of this class due to its structure, its composition is presented next.

The Class *ClsSystem* describes the system as a whole. It is defined as a set of agencies and hence inherits properties and methods from the class *ClsAgency* which is at level 3, as shown in Table 8.2. To read and comprehend the system, it is therefore required to know the class of agencies.

The class *ClsAgency* in turn, inherits two classes at the second level (the class of databases, *ClsDatabase* and the class of interfaces, *ClsIsales*), and two at the first level, (the class of global variables *ClsGlobalVaribles*, and the class of accounts *ClsAccount*), (see Table 8.2 for the hierarchical levels and Chapter 4 for the structuring of classes). To comprehend the class of agencies, it is also necessary to understand the above-mentioned four classes that the class *ClsAgency* inherits.

The class *ClsIsales* also inherits two classes: the class of accounts (*ClsAccount*) and the class of global variables (*ClsGlobalVariables*). The class of databases on the other side, inherits only the class of global variables (*ClsGlobalVariables*) and hence, appears to be more comprehensive. The class *ClsGlobalVariables* includes only basic types in its definition and no other class is inherited.

In Table 8.2, a class schema that inherits only from basic types is assigned the value 1 at level 1 of the inheritance and polymorphism irrespective of the number of basic types it uses. It is assumed that basic types do not bring difficulties in communicating a specification as they do not carry any information other than their name. Hence, the average number of classes that are not basic types at each level of inheritance is the following:

Level 1: 2 (two) classes (which represent 40% of classes) other than basic types.

Level 2: 2 (two) classes (representing 40% of classes) other than basic types.

Level 3: 1 (one) class (representing 20% of classes) other than basic types and.

Level 4: 1 (one) class (representing 20% of classes) other than basic types.

It appears that due to the inheritance, the analysis of the class *ClsSystem* requires the study of all the other classes of the system (see Chapter 4). This may not pose a great deal of difficulty when the number of classes involved is relatively small. However, the situation remains challenging because the reader of the specification needs to keep a picture of those classes in mind (according to conventional human thinking styles as discussed by Hatton [35]).

Also observe that the inheritance in most of the classes resulted from the Object-Z transformation of the Z schemas, used as types in abstract state schemas. For example, the state schema *Agency* (see Chapter 3, Section 3.4.2) defines the following component:

$$ssales : \ ISales \tag{8.6}$$

which is translated to

$$ssales : \ ClsISales \tag{8.7}$$

in the class *ClsAgency*, inducing the class *ClsAgency* to inherit properties and methods from the class *ClsISales*. It is important to consider this aspect, because the use of a schema in Z as a type, is problematic (van der Poll [90]). Although the intent is not to carry this analysis any further, one may however be concerned about the quality of an Object-Z class schema derived from an abstract state schema that utilises other Z schemas as types.

In the light of the above analysis, and further reasoning wherever necessary, conclusions on a number of properties of the specification are derived and presented in the upcoming paragraphs; starting with the *readability*.

Readability ($p1$)

From the above analysis based on Table 8.2, it appears that the reading of some classes of the Z-OZ specification is not straightforward. The reader is required to keep in mind the picture of one or more other classes due to the multiple hierarchical levels of inheritance involved in the structure of those classes.

It is also argued that the readability of a specification is further complicated by the way it is presented. Although it is common practice in Object-Z to have the definition of classes separated from their counterpart prose descriptions, as is the case with the Z-OZ specification, it may be observed that such a presentation forces a reader to go through a component of the class, without knowing what it is, and search for the relevant explanation in the prose

that may or may not follow the class. In this regard, the specification may need to be amended to improve upon the presentation and restructured to include, for example, only direct (one-level) inheritance in the specification. It is worth observing that this brings us to the maintainability ($p15$) and the modifiability ($p16$) of the specification that are addressed in the following paragraphs.

Next, the understandability of the Z-OZ Specification is investigated.

Understandability ($p2$)

In general, formal methods are said to have a steep learning curve, because of the use of mathematics for which practitioners are often not (sufficiently) trained, (Heitmeyer [37]). However, mathematical notations may not be the only source of difficulties. From the above discussion about the structuring of class schemas in an Object-Z specification, it can be observed that a multiple hierarchical level of inheritance and polymorphism may also contribute to render a specification hard to comprehend.

As mentioned for the readability, the Z-OZ specification under consideration needs to be amended in this regard, to render the whole system less complex and comprehensive.

Next, is discussed the *Clarity* property.

Clarity ($p3$)

As indicated by Duke et al. [27], the main reason for extending Z to Object-Z, was to improve the clarity of a specification through enhanced structuring. Therefore, the Z-OZ specification ought to be clearer than its Z counterpart from which it was generated. Each class schema in Z-OZ encapsulates a state schema, with all the operation schemas that may affect the state schema. However, as discussed for the readability, the presentation of the specification may be improved upon and hence, the readability and the clarity.

It may also be argued that restructuring the specification, as recommended for the understandability, would further improve the clarity. However, due to the duality between the clarity and brevity (as discussed by Gravell [34]), the clarity may be obtained at the cost of the brevity of the specification and hence, increase its size.

The following paragraph discusses the *Traceability* property.

Traceability ($p8$)

The aim is to map the Z-OZ specification under consideration to the initial requirements and stakeholders' expectations. Relating these three components of the system may help to detect, for instance, omitted requirements, or those that were changed during the specification process. It may equally help to detect pitfalls in the specification process.

For the Z-OZ specification, the initial goals $g1$, $g2$, and $g3$ (see Table 8.1) are not related to any specific components or objects in the specification, but instead to some qualities required from the specification as a whole. For example, as demonstrated earlier, $g1$ requires the specification to be readable, understandable, and clear (see formula 8.1 and Table 8.1). The lack of specific objects in the specification that describe those stakeholders' expectations, may stem from the fact that Z and Object-Z, do not explicitly describe non-functional requirements (see for example Dongmo and van der Poll [23], van der Poll and Kotzé [93]). For this reason, only three user requirements $g4$, $g5$, and $g6$, are formally related to the specification, in the form of a traceability matrix (Table 8.4). Such a relationship is also suggested by Formula 8.3.

To facilitate the understanding of the traceability matrix (Table 8.4), some short codes are proposed to reference users' requirements and class schemas in the Z-OZ specification in Table 8.3.

Requirements		**Z-OZ classes**	
g4	return item	cls0	*Basic Types*
g5	replace item	cls1	*ClsGlobalVariable*
g6	pay credit	cls2	*ClsAccount*
		cls3	*ClsIsales*
		cls4	*ClsDatabase*
		cls5	*ClsAgency*
		cls6	*ClsSystem*

Table 8.3: Table of codes

In Table 8.3, the first two columns are used to code the requirements, and the last two for the Z-OZ classes. Since basic types exist to be used by other classes, for brevity, we do not list them individually. A unique class code ($cls0$) is used to denote each basic type. Although the use of this code seems to be ambiguous, in the present situation, it may not be harmful because there is no need to reference each of the classes individually.

The traceability matrix is presented in Table 8.4.

Requirements	**Z-OZ Classes**						
	cls0	cls1	cls2	cls3	cls4	cls5	cls6
g4	x	x	x	x	x	x	x
g5	x	x	x	x	x	x	x
g6	x	x	x	x	x	x	x

Table 8.4: Traceability matrix

A cross in a cell in Table 8.4 indicates that the Object-Z class in the column is related to the requirement in the line. Such a relationship is accomplished when the definition of the class includes, at least one component or an operation that aims to partially or fully achieve the corresponding requirement. For each user requirement, at least one basic type ($Cls0$) is used in the definition of one or more classes that achieve the requirement hence, relating the class of basic types to each and every user requirement. E.g. when returning or replacing an item, the class of interface of communication, denoted by $cls3$, is used to refund a customer or to specify a warning message ($UnknownCustomer$), to alert a user that the customer does not have a valid account with the local company. Those two operations are indicated by the following expression:

$$RefundCust \mathrel{\widehat{=}} RefundCustOk \parallel UnknownCust \tag{8.8}$$

Similarly, when making a payment, the operation $UnknownCust$ of the class of interface of communication, $ClsIsales$, denoted by $cls3$, is inherited in the class of agencies (denoted by $cls5$), to model a warning message ($UnknownCust$), whenever a customer does not have an account with the local company. This claim is justified by the following formula included in the class of agencies ($cls5$):

$$\begin{aligned} ReceivCash \mathrel{\widehat{=}} & receivCashOk \parallel UnknownIdentifier \parallel AgencyNotFound \\ & \parallel UnknownCust \end{aligned} \tag{8.9}$$

The operation $UnknownCust$ is automatically selected by the system to check if a customer has a valid account. When the customer does not have a valid account, a warning message is generated and the user is alerted. The operation is accessed through the component $ssales$, which is an inherited object of the class denoted by $cls3$. Formula 8.9 may become clearer if the name of the operation, within the expression is changed to $ssales.UnknownCust$ that clearly indicates the object from which the operation originates.

An analysis of the relationship between users' requirements and the Z-OZ class schemas

can also be undertaken, since Table 8.4 is amenable to a Parnas table (Parnas [67]). In this regard, crosses in the cells may be replaced by the expressions (including variables and predicates) in classes that relate those classes, to user requirements.

In the light of the traceability matrix in Table 8.4, the following observations may be made:

(1) Each user requirement is related to each and every class schema within the specification.

The above observation suggests that the accomplishment of each of the listed requirements (in Table 8.4), necessitates the availability of the entire system at once. Consequently, this may compromise the quality of the envisioned system because of the following reasons:

Reliability: If a single component within the system is faulty, the entire system may become unusable.

Maintainability: The system, therefore, becomes hard to maintain, as any faulty object may bring the entire system down.

Security: All the system components must be equally secure.

Performance: Having all the components available at the same time may render the system sluggish and reduce the performance.

Complexity: As in the case of the performance, the achievement of a single user need may require lot of resources because of the large number of objects needed.

This situation may also be an indication of redundancies in user requirements, which may be attributed to the fact that the initial users' requirements were not transformed into system (or software) requirements (Sommerville [82]), before specification. To illustrate the claim about refining initial requirements to generate system functionalities, consider for example breaking-down the requirements $g4$ and $g5$ as followes:

Requirements		**sub-requirements**	
g4	return item	g41	receive an item from a customer
		g42	send item to agency
		g43	refund a customer
		g41	receive an item from a customer

g5	replace purchased item	g42	send item to agency
		g51	send a different item back to customer

Table 8.5: Requirements and sub-requirements

Table 8.5 clearly shows that some sub-activities needed to accomplish $g4$ and $g5$ are duplicated. Further transformation of the sub-activities may be performed until appropriate system requirements are obtained. Such a transformation may help to discover and eliminate possible inconsistencies in initial requirements.

The above illustration may be perceived as an indicator that a separate requirements analysis phase (including at least sketching the architectural design of the system) must precede the formal specification of the system, as suggested by Sommerville [82], in Figure 10.1. Such an analysis phase may aim not only to transform user requirements into system requirements, but also to architecturally reveal a possible conceptual relationship between them. In line with this idea, van Lamsweerde [95] states:

Formal specification is not a mere translation from informal to formal. The specification of a large, complex system requires relevant objects and phenomena to be identified, interrelated, and characterized through properties of interest.

(2) As shown in the traceability matrix (Table 8.4), having each component defined in the specification contributing to the achievement of all the user requirements may be an indicator of poor structuring of the specification.

The poor structuring of the specification was mentioned in the previous paragraphs when analysing the nested inheritance. It has just been demonstrated that such a problem could also be addressed by transforming the user requirements into system requirements and sketching the architectural design, before proceeding to the formalisation of the specification. This proposition may be valuable, considering that Z as a specification technique, does not include any mechanism to address the architectural aspect of a system in the first place (see e.g. Dongmo and van der Poll [23]). It also fails to explicitly address the problem of providing mechanisms to capture user requirements and build system specifications from them, for example, in an incremental way (see e.g. van der Poll and Kotzé [93], van Lamsweerde [95]).

Completeness ($p9$)

It may be hard to objectively decide on the completeness of the specification since, for example, as mentioned above, some qualities expected from the system are not traceable. However, the traceability matrix indicates that all the user requirements are addressed by the specification. Therefore, at this stage of validation, it is revealed that the specification under consideration, embodies functionalities that fulfill the users' requirements. However, more analysis is needed to establish whether those functionalities adequately accomplish such requirements.

Operational feasibility ($p4$)

Balzer and Goldman [10] discussed as early as 1981, in their *principle* 6, the importance of having a specification operational. In the context of this dissertation, by dealing with this property, the aim is not to conduct the operational study itself but, to evaluate how well the specification under consideration may facilitate such a study at the current stage. Although the intention is not to do a deeper analysis, it may be worth looking, for example, at how easily complete scenarios, can be reconstructed from the specification, and the degree of detail and simplicity in which the operations are described. Those two aspects may help, for instance, to determine user satisfaction. The reason is that having complete scenarios together with simple, detailed and understandable operations, may, for example, facilitate the process of identifying the position and responsibilities of potential users in the scenarios. Hence, one would therefore be prepared to analyse and discuss their satisfaction, either empirically or theoretically.

To illustrate, consider the following scenario that describes the process of a cutomer returning an item to the provider via a local agency.

1. A customer returns an item to a local agency say, AgencyA which is not the provider of the item.
2. The item is received from the customer.
3. The item is temporarily kept in a store, pending a transfer to the provider.
4. The item is sent to the provider, say AgencyB.
5. AgencyB updates the customer account and notifies the customer about the transaction.

An attempt to reconstruct this scenario from the specification, results in the following Z schema composition expression in which *system* is an instance of the class *ClsSystem*.

$$system.this.ReceivItem \fatsemi system.sendItem \fatsemi system.NotifyCustTrans \tag{8.10}$$

Building formulae like (8.10) for large (and complex) systems can be hard if there is no initial requirements capturing and analysis strategy. The standard Z specification technique does not explicitly include the concept of scenarios. It focuses mainly on defining system states and operations. To construct formula 8.10, classes were inspected aiming to identify any operation that may be involved in the scenario. The formula itself represents a Z schema composition expression. Although it clearly indicates the sequence of operations required for the scenario, it may not succeed in facilitating the operational feasibility study. The reason is that it does not explicitly describe the distant communication between the two agencies involved in the process.

The three operations in formula 8.10 are very compressed and hence, hide complex activities in the specification. To illustrate this, consider the operation *receivItemOk*, which is internally applicable when the pre-condition of *ReceivItem* is satisfied. The predicate part of the operation includes the following expression:

$$
\begin{aligned}
&\exists\, Agency \bullet \\
&\qquad (\\
&\qquad \theta Agency.identifier = id? \land && (8.11)\\
&\qquad inv? \in \theta Agency.ssales.invoices \land && (8.12)\\
&\qquad addr! = \theta Agency.address \land lang! = \theta Agency.language && (8.13)\\
&\qquad)
\end{aligned}
$$

The following activities are encoded by the above expression:

1. The identification of the company for which the identifier is known ($id?$) as input (8.11).
2. (Remotely) request the company to validate, via its communication interface, the invoice that was provided by the customer (8.12).
3. (Remotely) request the address of the company and the language to use for communication (8.13).

It appears that each of the above enumerated tasks is highly dependent on a network communication or broadly, a distant communication between the companies involved in the operation. However, the specification itself might not make this aspect of the system sufficiently obvious to the reader. Consequently, if for example, at the design and implementation phases, the designer does not apprehend the unforseen impact of such a communication, provision may not be made, for example, for networking (installation, administration, communication protocol, telephone system, etc.) in an economic efficient and operational basis. Having

many different activities compressed in a single formula may also introduce the problem of cohesion (Zhao and Xu [103]) in the system.

Next is presented the leftward validation phase.

8.3.3 The leftward validation

Creativity ($p7$)

As described in Section 3.1, (Chapter 3), stakeholders initially intended to increase profit in their business. Suppose the idea was not based on any existing business from which processes and operations could be derived. However, the operational environment is already known (an interconnection of different, well-known companies physically dispersed world-wide). This renders the system highly reliant on the ability of the specifier to analyse the environment, and propose appropriate operations to specify system functionalities. In this regard, the property of *creativity* is used in this dissertation (as a means to observe the ability of the specification technique or process to stimulate the specifier's imagination about the required functionalities of the system).

Although a problem like the one that has just been discussed in the previous section, about the operational feasibility, may be attributed to the specifier's lack of skill, one may equally see it as a weakness of the specification process and the technique used. It may be observed that when constructing the Z and Object-Z specification of the scenario, the aim was to accurately describe operations that were identified by analysing the user requirements and provide as much detail as to clearly determine the circumstances under which an operation will succeed, or fail. For example, a verb or verb phrase in a sentence is likely to lead to an operation (see van der Poll [90]). Preconditions were calculated for each operation in the Z version of the specification. While the specification technique indeed provides very powerful mathematical tools to adequately code identified operations, it remains largely silent about helping the specifier to address, for example, the processes, scenarios, the degree of abstraction, and the cohesion of operations.

The main focus of the enhanced strategy for constructing Z documents is to concentrate on identifying, from the statement of the user requirements, those objects that will serve to best describe the abstract states and the operations of the system. Some heuristics were suggested by van der Poll and Kotzé [93] to reinforce the strategy that recommends amongst others to have operations decomposed into primitives, with the main purpose to facilitate automated proof, but also to facilitate cohesion. However, the main problem of addressing

system processes and scenarios, and bringing operations to a reasonable level of abstraction before a specification, still remains. Hence, there is a need for a complementary tool at the level of user-requirements. In line with these observations, Sommerville [82] suggested the use of formal specification techniques after user requirements have been captured, analysed and transformed into system requirements.

Networking ($p10$)

Neither this property, nor the **creativity** discussed in the previous paragraphs, are standardised properties (see for example IEEE Std 830-1998 [41]) that are commonly used to characterise or judge the quality of a specification. The networking property is related to the nature of the application domain, which involves an intensive communication between companies geographically dispersed world-wide (see Figure 3.1, Chapter 3). The aim is to observe how well the specification technique or process may assist, for example, a requirements engineer to focus on such aspects of an innovative system, where detailed descriptions of user requirements have to be deduced or discovered by observing, the application domain.

The above discussion on the **operational feasibility** and the **creativity**, clearly indicates that network communication is not explicitly handled by the proposed Z-OZ specification. One possible explanation is that operations in the specification are described at the same abstraction level as user requirements, whereas some authors (e.g. Sommerville [82]), suggest that formal methods be applied at a lower level of abstraction, to system requirements rather than user requirements. Also, Z and Object-Z provide very powerful mathematical tools to help a specifier explore the circumstances that may surround an operation but, do not provide mechanism to reason about full processes or scenarios. These are the elements from which the possible inter-dependencies between operations can be observed facilitating the analysis of any interaction between system components and possibly discovering missing operations.

Integrity ($p11$)

This property aims to analyse the integrity of the specified functionalities within the application domain. Since standard Z, from which the Z-OZ specification was generated, does not explicitly provide means to reason about scenarios, the point is to ensure that each operation fits adequately into at least one complete process or scenario. One way to fulfill this aim, would be to identify, from the operational environment, all the processes that interact or operate in isolation, to build the entire system.

Table 8.4 shows that each class in the specification contributes to the achievement of at least one of the known user-requirements. As operations are encapsulated into classes, it appears that operations themselves contribute to achieving user requirements. If at this level, it is assumed that each complete scenario may be generated from user requirements, hence, it may be deduced that each functionality in the specification is, clearly, part of a scenario. However, further investigation may be conducted to determine the adequacy or appropriateness of such functionalities, e.g., by animating the specification (e.g. McComb and Smith [56]).

The following paragraphs discuss the rightward validation phase of the Z-OZ specification.

8.3.4 The rightward validation

In practice, this phase of the validation process mainly involves the use of tool support to, for example, to perform the syntax checking, and the semantic analysis of the specification. The use of such tools may equally assist the specifier in discharging some proof obligations.

Correctness ($p12$)

With respect to the language notation, the concern is to ensure that the specification is syntactically and semantically correct. The Z and Object specifications presented in this document were generated using the OZ.Sty (Allen [3]) package of Latex macros for printing Z and Object-Z specifications. The tool provides macros for Z and Object-Z fonts, including symbols and boxes. The standard Latex syntax checking is performed when compiling the document. However, the Z and Object-Z related syntax and semantic analysis are not supported.

As observed above, to infer an accurate conclusion about the correctness of the specification with respect to its encoding, tools for syntax checking, and semantic analysis, are also required. As reported in Dongmo and van der Poll [24], a number of tools exist for Z and Object-Z, and many of them run under the Linux system. For example, the Community Z Tools CZT for editing, type-checking and animating Z and Object-Z specifications(see Malik and Utting [54]). The Wizard: a type-checker for Object-Z in Latex (see Johnston [43]), Moby/OZ, a graphical editor to build Z and Object-Z specifications. To date, detail information on tool support for Z and Object-Z, as well as, publications are accessible via a number of formal methods Wiki web sites: (e.g. `http://formalmethods.wikia.com/wiki/Z_notation`). Some research has suggested the encoding of Object-Z, into existing automated theorem provers (see for example Smith et al. [80]), and the re-use of a Z animator for Object-Z (e.g.

McComb and Smith [56]).

The purpose here is not to utilise those tools to analyse the Z-OZ specification at hand, but rather bring attention to their existence. However, some considerations need to be mentioned regarding their applicability. Although most of the existing tools are Latex based, the Object-Z specification at hand needs to be updated to render it applicable. For some (e.g. the Wizard, Moby/OZ, etc.), only very slight modifications may be required. For example, to type-check the Z-OZ specification with the Wizard type-checker, the following updates, which are not exhaustive, are necessary within the Latex source document:

1. Include the package wizard.sty to the document.
2. Separate declarations, with a semicolon (;) or an **eol**.
3. Predicates need to be separated by an **eol** or a semicolon (;).

Next aspects related to the internal consistency are discussed.

Internal consistency ($P13$)

It is argued that having a consistent requirements specification is a critical quality on which the correctness of an implemented system relies (IEEE Std 830-1998 [41], Liu [53]). However, addressing the consistency of a specification can be a tedious task, since it involves demonstrating that the specification is semantically sound (Sommerville [82]), that is, it does not contain any contradictions. For large and complex systems, manual consistency checking can be time-consuming and hence, very costly (Heitmeyer et al. [38]). To investigate the internal consistency of the Z-OZ specification, a twofold manual approach is adopted in this dissertation. Firstly, each class schema is analysed, individually aiming to reveal any contradictions within the class. Secondly, the system is investigated as a whole, to identify possible inconsistencies between classes due, for instance, to over-lapping or inter-related components in the classes.

The result of the manual investigation is presented next.

1. No error was found for the class of basic types, nor the class of global variables (*ClsGlobalVariable*), nor the class of accounts (*ClsAccount*).
2. Within the class of Interface Communication (*ClsIsales*), about five errors were found, but three of which are syntax errors, that could be eliminated by applying an automated tool support, (namely, a type-checker), as suggested in the previous paragraphs. The

other two are found in the following expression of the operation *refundCustOk*:

$$\exists\, Account \bullet \tag{8.14}$$
$$($$
$$\theta Account \mapsto cust? \in custaccounts \tag{8.15}$$
$$statements' = statements \cup (\theta Account, trans) \mapsto date? \tag{8.16}$$
$$)$$

The term *Account* in Formula 8.14 refers to a component that does not exist in the specification. This error induces the other two expressions 8.15 and 8.16 to produce errors. *Account* is an abstract state in the Z specification, from which the Object-Z specification under consideration was generated. So, the error may be due to a confusion between concepts in Z, and its counterpart in Object-Z, or a careless application of the transformation guidelines. For example, by copying parts of the Z specification and pasting it into an Object-Z class without ensuring a proper transformation, or by forgetting to do so afterwards.

3. Within the class of databases (*ClsDatabase*), three errors were found. One typing error, and the use of the Z abstract state space *Account* (the same as discussed above) that induces the other two errors as shown in formulas 8.17 and 8.18.

$$\forall\, Account \mid \theta Account \in \mathrm{dom}(statements)) \bullet \tag{8.17}$$
$$($$
$$\exists\, id : Agencyid \bullet id \in \mathrm{ran}(cashin) \wedge$$
$$\theta Account \mapsto id \in agencyaccounts \tag{8.18}$$
$$)$$

4. Within the class of agencies (*ClsAgency*), about seven errors were found. Three of these are referencing errors, and the other four concern the use of Z abstract state schema as objects in the Object-Z, without transformation. One example of the referencing error is the use of the component *agencies* in the predicate part of the class.

$$identifier \notin agencies \tag{8.19}$$

The component *agencies* is inherited from the class of databases, for which an instance is accessible in the class of agencies through *db*, that represents a pointer to a database object. Therefore, a mere reference to *agencies*, in the class of agencies should instead use the notation *db.agencies*. Similar observations are made for *agencyaccounts*

in the predicate part, and the component *itemsin* in the predicate part of the operation *receivItemOk*. Other types of errors, stem from the use of the Z schema *Agency*, as a type in the specification of the operations, *receivItemOk*, *AgencyNotFound* and *InvalidInvoice*.

5. Consider the two components of the class *ClsSystem*, defined as follows:

$$this : ClsAgency \tag{8.20}$$

$$known : \mathbb{P}\, ClsAgency \tag{8.21}$$

A closer analysis of these two components reveals that the state schema of the class system is redundant, as the same functionality is specified within the class of database. In fact, the idea is of a decentralised system, where each agency keeps its own database. The component *known*, represents all the agencies participating to the system, and *this* is the pointer to an object of the class of agencies, that designates the agency in which an instance of the system is operating. Consider also formulas 8.22 and 8.23 (see Section 4.2.2 in Chapter 4) defined, respectively, in the state schema of the class of agencies, and that of databases.

$$identifier : Agencyid \tag{8.22}$$

$$agencies : \mathbb{P}\, Agencyid \tag{8.23}$$

Identifier is the identifier of the agency in which the system is operating, and *agencies*, the list of agency identifiers. Formula 8.22 specifies the same idea as formula 8.20, with the difference being that the component (*this*), references a larger object that eventually contains the component *identifier*. Similarly, formulae 8.21 and 8.23 serve the same purpose within the same system, with the difference being that an object in the list *known* has more components, than that of the list *agencies*. The list *agencies* is inherited in an object of the class system, through the promotional expression: *this.db.agencies*, meaning that the two lists are managed concurrently within the same instance of the class system.

The errors identified so far are not exhaustive, since further analysis could be done to address functions and processes, for example, the consistency between post-conditions and input data availability, consistency between pre- and post-conditions and eventually, consistency between processes (Liu [53]).

Minimality ($P14$)

It is suggested that a good software specification should be as minimal as possible (van Lamsweerde [95]). To ensure this characteristic, a specification is required to include only

properties that are relevant to the problem or the problem solution. The traceability matrix in Table 8.4, shows that each class schema in the specification is related to a user requirement. Further analysis of individual classes does not reveal any components or operations that are irrelevant to the system.

Maintainability ($P15$)

An acceptable software requirements specification should be modifiable to facilitate its maintenance (see Balzer and Goldman [10], IEEE Std 830-1998 [41]). The modification of a software specification may become easier, if the structure and style of the specification is made simple and easily understandable. It was discussed previously that the structure of the Z-OZ specification under consideration, may be suspect because the standard Z specification technique, from which the specification was derived, does not support system architectures in the first place. Another reason discussed earlier, is the use of nested inheritance, that may also render the structure of the specification complex, and hence, hard to understand.

Next, the downward validation phase is discussed.

8.3.5 The downward validation

Since the purpose of this validation phase is to demonstrate that the specification under consideration is likely to lead to a software product with the required qualities, the technical feasibility and the executability of the specification are now addressed.

Technical feasibility

As pointed out by McConnell [57], a requirements specification of a software system may be a valuable tool to guide the technical feasibility of the system. Since with a specification one aims amongst other objectives, to produce a quality software product, the transformability of the specification, becomes an interesting aspect to focus on. The issue may be, for example, to check if there exist appropriate techniques or methods to transform the Z-OZ specification, into operational software products. To this end, as reported in Dongmo and van der Poll [24], a number of refinement methods have been suggested for Object-Z (e.g. Duke and Rose [26], Qin and He [71], Smith and Derrick [79]). Johnston and Rose [44] also suggest guidelines to implement an Object-Z specification with the C++ programming language, (and some practical projects with Object-Z are reported on the Object-Z website: `http://www.itee.uq.edu.au/smith/publications.html#tools`.)

Owing to the above observations about the transformation of an Object-Z specification, the

aim is no longer on investigating if the specification can be transformed or not, but rather focuses on the ability of the specification to facilitate such a transformation. Amongst other properties, the ability of a specification to be refined or transformed into a final product, highlights the modularity / structuring, clarity, understandability and readability of specifications, discussed previously. Since it is believed that a specification that encompasses those qualities may be easy to communicate to team developers, facilitating the application of transformation techniques as developers clearly understand the system to be produced. Another important aspect to focus on, is the executability of the specification, which is discussed next.

Executability

As early as 1992, Fuchs [32] suggested a software specification should be executable, as it would exhibit the behavioural aspect of the envisioned software, during the early conception phase. Prototyping has been used as a means to demonstrate the feasibility of a proposed system and demonstrate its behaviour. However, the technique requires the partial design and implementation of a system. With formal techniques, automated proofs and animations are meaningful ways to focus when demonstrating certain qualities of a specification, without proceeding with further development phases. A number of animation strategies have been proposed for Object-Z such as: animating Object-Z using Z animator (McComb and Smith [56]), encoding Object-Z in Isabelle/HOL (Smith et al. [80]), and An XML/XSL Approach to Visualize and Animate TCOZ[1] (Sun et al. [85]).

8.4 Validating the UCM-OZ specification

This section applies the validation framework, developed in Chapter 7, to validate the UCM-OZ specification of the case study presented in Chapter 6. As recommended by the framework, the process that was followed to construct the specification, is briefly presented in Section 8.4.1. Since the aim of the validation process in sections 8.3 and 8.4 is to enable the comparison of the two specifications, Z-OZ and UCM-OZ, in the light of the discussion from Section 8.3, this section does not insist on properties that the two specifications may have in common. For example, the specifications specify the same set of user-requirements, use concepts and terminologies of Z and Object-Z. Instead, the focus is on those aspects of the UCM-OZ specification that may highlight the impact of the Use Case Map specification techniques introduced into the process, to generate the final specification.

[1]TCOZ (Timed Communicating Object-Z) is an integrated formal notation that build on Object-Z's strengths in modeling complex data structures, and on Timed CSP's strengths in modeling real-time interactions

8.4.1 The UCM-OZ specification

As in the case of the Z-OZ, Figure 8.2 illustrates the three- step process followed to construct the UCM-OZ specification.

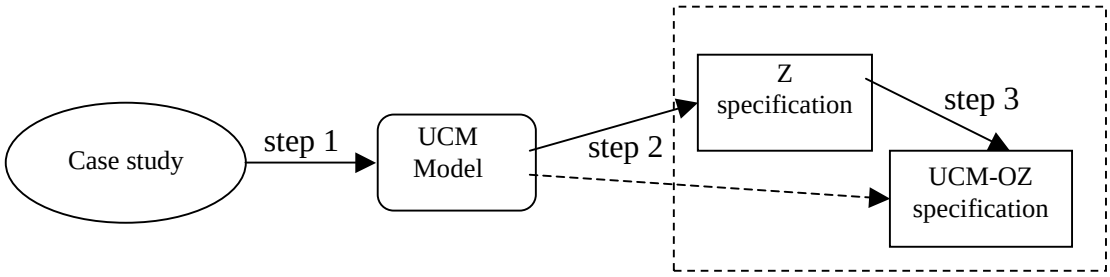


Figure 8.2: UCM-OZ specification process

The case study was described in Chapter 3. The first step transforms the case study into a UCM model (see Chapter 3). Then, during the second step, the framework developed in Chapter 5, is used in Chapter 6, to guide the transformation of the UCM model, into a Z specification, and to generate meta-classes for Object-Z, and then, in step 3, to Object-Z. Step 3 in this process is identical to step 2 in the two-step process of the Z-OZ specification discussed in Section 8.1. During this last step, the Z specification and meta-classes are combined, to form Object-Z classes.

Before proceeding with the upward validation phase in the following paragraphs, the list of those UCM-OZ class schemas on which the discussion is based, is presented first. The classes were described in Chapter 6. Table 8.6 presents the list of classes in two columns.

UCM-OZ classes		**UCM-OZ classes**	
cls0	Basic Types	cls11	ClsCashier
cls1	ClsGlobalVariables	cls12	ClsPayPoint
cls2	ClsTimer	cls13	ClsInStore
cls3	ClsInterface	cls14	ClsTransitPoint
cls4	ClsNetInterface	cls15	ClsHelper
cls5	ClsNetComTemp	cls16	ClsCheckPoint
cls6	ClsRequest	cls17	ClsUpdatePoint
cls7	ClsLocalSales	cls18	ClsBeneficiary
cls8	ClsCheckInvoice	cls19	ClsMainStartPoint
cls9	ClsCheckCustomer		
cls10	ClsInitChecking		

Table 8.6: List of selected UCM-OZ Classes

The following section discusses the upward validation phase where the focus is on the properties revealing the ability of the specification to satisfy stakeholder expectations.

8.4.2 The upward validation

For similar reasons presented in Section 8.3.2, the properties to check for the UCM-OZ specification are: the readability ($p1$), the understandability ($p2$), the clarity ($p3$), the operational feasibility ($p4$), the traceability ($p8$) and the completeness ($p9$).

Table 8.7 aims to reveal the complexity of the UCM-OZ specification, relative to the structuring, due to nested inheritance and polymorphism.

No.	Class	**Inheritance**					**Polymorphism**		
		1	2	3	4	5	1	2	3
00	Basic types	0	0	0	0	0	0	0	0
01	ClsGlobalVariables	1	0	0	0	0	0	0	0
02	ClsMainStartPoint	1	0	0	0	0	0	0	0
03	ClsTimer	1	0	0	0	0	0	0	0
04	ClsInterface	1	0	0	0	0	0	0	0
05	ClsNetInterface	1	1	0	0	0	0	0	0
06	ClsNetComTemp	1	0	1	0	0	0	0	0
07	ClsRequest	1	4	0	1	0	0	0	0
08	ClsLocalSales	1	0	0	0	0	0	0	0
09	ClsCheckInvoice	1	1	0	0	0	0	0	0
10	ClsCheckCustomer	1	1	0	0	0	0	0	0
11	ClsInitChecking	1	2	0	0	0	0	0	0
12	ClsCashier	1	0	0	0	0	0	0	0
13	ClsAgenciesAccounts	1	0	0	0	0	0	0	0
14	ClsPayPoint	1	3	0	0	0	0	0	0
15	ClsInStore	1	0	0	0	0	0	0	0
16	ClsTransitPoint	1	1	0	0	0	0	0	0
17	ClsHelper	1	0	3	0	1	0	0	0
18	ClsCheckPoint	1	1	3	0	0	0	0	0
19	ClsUpdatePoint	1	1	0	0	0	0	0	0
20	ClsBeneficiary	1	1	1	1	0	0	0	0

Table 8.7: Levels of inheritance and polymorphism in UCM-OZ

The indicates that the UCM-OZ specification includes a total of twenty (20) class schemas. Next the information on each level of inheritance, is revisited where the number of classes and the average number inherited classes per lower level, is also provided.

Level 1: Eight(8) classes, representing 40% of the total number of classes. These classes use in their definitions only basic types.

Level 2: 7 Seven (7) classes, representing 35% of all the classes. Excluding basic types, on average, each of these classes inherits about $10/7 = 1.4$ classes at level 1.

Level 3: Two (2) classes, representing 10% of all the classes. Excluding basic types, on average, each of these two inherits about half (1/2) a class at level 1 and 2 classes from level 2.

Level 4: Two (2) classes, representing 10% of all the classes. Each of these classes inherits on average, 5/2=2.5 classes from level 1, half (1/2) a class from level 2, and one (2/2) class from level 3.

Level 5: One (1) class, representing 5% of all the classes. The class inherits three classes at level 2, and one at level 1.

The table also indicates that polymorphism is entirely absent from the specification. However, it is valuable to observe that Object-Z classes, resulting from the transformation of a dynamic stub, (see e.g. Figure 6.2), may specify polymorphic objects. For example, by combining the two class *ClsCheckInvoice* and *ClsCheckCustomer*, to form a class that inherits both of them, and includes in its interface the visibility list of both classes. An object of such a class, can either be used to check an invoice, or to check a customer.

The following paragraph considers the readability of the specification.

Readability ($p1$)

From the above observation on the number of classes in each hierarchical level of inheritance based on Table 8.7, it appears that the reading of most (about 75%) of the UCM-OZ specification class schemas, may not be difficult to the average person who is reasonably familiar

with Object-Z terminologies and concepts. A small proportion (about 10%) of the specification would require more effort from the reader to understand two or three classes. Hence, a tiny part (15%) of the specification represents the hardest part to read and comprehend, as it requires a reader to study from 1 to 4 other classes in sequence. It is, however, important to notice, by looking for example, at the space that each class occupies on paper, that the average size of each class is manageable. The size of those classes may have a positive impact on the readability of the specification, since a reader would not need too much efforts to read and comprehend one class. Nevertheless, including the short prose text inside each class, for example, to explain a component within the class, may help render a specification more readable.

Next the understandability of the UCM-OZ specification is discussed.

Understandability ($p2$)

As mentioned earlier when discussing the understandability of the Z-OZ specification, having multiple levels of nested inheritances in a specification, renders the specification hard to comprehend. However, with the UCM-OZ specification, a valuable aspect is the support that the UCM model provides to facilitate the understanding of the specification. As shown in Figure 8.2, each class or a sub-set of classes in the UCM-OZ is clearly a result of the transformation of a UCM component, or a sub-map. The operations in a class are those previously defined in a UCM. Unlike the Z-OZ specification, where most of the nested inheritances are induced by the use of schemas as types, the structure of the UCM-OZ specification, is based on the architecture of the components, within the UCM model. This observation implies that both the comprehension of individual elements within the UCM-OZ class, and that of the entire specification, may be readily available through the UCM elements from which it is generated.

The next section discusses the clarity of the specification.

Clarity ($p3$)

It is argued that a specification is clearer when it is well structured, with each element having a reasonable size. For example, regarding the size, a class schema with a large number of variables and operations, that occupies more than two pages of an A4 document, might be confusing to the reader. Hence, since Object-Z specifications are inherently assumed to be clear, because they tend to be modular (Smith [77]), the UCM-OZ specification may be viewed as being clear, due to the concise size of its class schemas. It also owes its clarity to its structure, that can be assessed in figures 6.6 and 6.7, and the architectural structuring

of the UCM model (see Figure 6.2) components, on which they are based.

The ability to trace user requirements through different stages of a software development process, is one of the important qualities expected from a process. Next, the extent to which elements in the UCM-OZ specification, can be traced back to the initial user requirements, is discussed .

Traceability ($p8$)

Recalling the UCM model of the case study constitutes a step towards analysing the initial user requirements, in which each component assumes some specific responsibilities; two stages are considered to analyse the traceability. The first relates initial user requirements to UCM components (shown in figures 6.2, 6.3, 6.4 and 6.5), and the second relates UCM components, to Object-Z Classes.

Table 8.8 represents the traceability matrix, relating the UCM components in the UCM model of the case study, and the three initial user requirements, presented in Table 8.3.

UCM Components	**User requirements**		
	g4	g5	g6
Initchecking	x	x	x
NetControl plugin	x	x	x
CheckInvoice plugin	x	x	
CheckCustomer plugin	x	x	x
Cashier			x
Pay_point	x		x
Store	x	x	
Transit_point	x	x	
Helper	x	x	x
Check_point	x	x	x
Update_point	x	x	x
Beneficiary	x	x	x
Network	x	x	x

Table 8.8: Traceability matrix relating UCM components to users requirements

A relationship between a component and a requirement is identified by following, e.g., a path traversing the map, from a start-point to the end on the stubbed map, in Figure 6.2. For example, with a UCM, a scenario to return an item (g4) is modelled with a path, starting form the start-point $S1$, traversing components, and ending at the end-point $E1$. A component traversed by such a path is therefore related to the requirement $g4$. When a path gets into a static stub, the plug-in connected to the stub, is also related to the requirement, whereas, when it gets into a dynamic stub, at least one of the plug-ins associated to the stub ought to be considered; the selection policy to select those plug-ins may be used as a guide.

The traceability matrix, relating each class schema to a UCM element, or sub-map is presented in Table 8.9. The main purpose of the two matrices in tables 8.8 and 8.9, is to provide a means to trace back from the specification, to the user requirements, aiming to facilitate the analysis of both the specification, and the specification process.

	UCM Elements												
Class code	InitChecking	NetControl	CheckInvoice	CheckCustomer	Cashier	Pay_Point	Store	Transit_Point	Helper	Check_Point	Update_Point	Beneficiary	Network
Basic types	x	x	x	x	x	x	x	x	x	x	x	x	x
ClsGlobalVariables	x	x		x		x				x			x
ClsTimer		x											
ClsInterface		x											x
ClsNetInterface		x											x
ClsNetComTemp		x											
ClsRequest		x											x
ClsLocalSales			x	x							x		
ClsCheckInvoice			x							x			
ClsCheckCustomer				x						x			
ClsInitChecking	x								x				
ClsCashier					x	x							
ClsAgenciesAccounts						x							
ClsPayPoint						x			x				
ClsInStore							x	x					
ClsTransitPoint								x	x				
ClsHelper									x				

ClsCheckPoint										x		x	
ClsUpdatePoint											x	x	
ClsBeneficiary												x	

Table 8.9: Traceability matrix relating UCM-OZ to UCM components

To avoid having to list each class of basic types in Table 8.9, the expression **Basic types** is used as a generic expression to designate each individual class of such types.

A class schema is related to a UCM element, if the class was generated directly from the element (for example as a meta-class) during the transformation process (presented in Chapter 6), or if it is inherited from a class that was generated from the element. For example, at least one basic type, is needed to specify each UCM element.

Along a row, the matrix conveys information about the list of UCM elements to which a class is related. For example, the class *ClsCashier* contributes to the specification of the UCM object named *Cashier*, and the UCM component *Pay_point*, whereas along a column, two types of information are provided: first, the list of all the classes that together contribute to the specification of the UCM element in the column. For example, the plug-in *CheckInvoice* (shown in Figure 6.4), in addition to basic types, is completely specified with two Object-Z classes: *ClsLocalSales*, and *ClsCheckInvoice*. Secondly, it gives an idea of a relationship between classes. For instance, in the previous examples, as the system that UCM-OZ specifies, may not be allowed direct access to the local sales system of a company, the class *ClsCheckInvoices*, which checks invoices that are managed by the local sales system, needs to request an appropriate service, via the communication interface, specified by the class *ClsLocalSales*.

From the two tables 8.8 and 8.9, it becomes easier to relate class schemas to the user requirements, because the mapping between Object-Z classes, and the UCM elements on one side, and those between the elements of UCMs and the user requirements, are readily available. For example, to identify the list of classes that specifies the user requirement $g6$, the process is the following: from Table 8.8, the list of all the UCM elements related to $g6$ is identified. Then in Table 8.9, the list of all the classes that specifies those elements is selected. An important advantage of using the UCM technique in the process of constructing the specification is that, with the support of the UCM model, reconstructing a scenario

from the selected list of classes is relatively easier, because the UCM technique is inherently scenario-based. With a UCM model, a complete scenario is modelled with a (UCM) path, that begins from a start-point, traverses UCM elements, and ends with an end-point. The Class schema *ClsHelper* readily specifies the part of such paths that traverse the component *Helper*, and the class schema *ClsBeneficiary*, specifies those elements of paths that traverse the component, say *Beneficiary* (Figure 6.2).

Next the completeness of the UCM-OZ specification is discussed.

Completeness ($p9$)

By observing Table 8.8, it is clear that the UCM model provides a list of components to address each of the three user requirements $g4$, $g5$ and $g6$. It is also evident from Table 8.9 that each of the elements is specified with an appropriate set of class schemas. Moreover, the above discussion about the traceability of the UCM-OZ specification, indicates that a scenario describing the steps towards achieving any of the users' requirements, can be re-constructed from the specification. Based on these observations, the completeness of the specification relative to user requirements may be concluded. However, it is important to notice that such a conclusion does not go beyond the three listed user requirements, as other aspects of the system, such as non-functional requirements, are not considered.

The discussion of the property of operational feasibility follows.

Operational feasibility ($p4$)

As indicated in the discussion about the Operational feasibility of the Z-OZ specification on page 175, the aim is not to conduct an operational feasibility study of the system, in this dissertation, but to evaluate the contribution of the specification towards facilitating such a study. In this regard, the specification is expected to provide a clearer understanding of a scenario, and specify operations clearly. The above discussions on the traceability and the completeness, show how the UCM-OZ readily integrates the description of scenarios in its inheriting classes. Two examples are given to illustrate this.

Example 1: The class schema *ClsRequest* (Chapter 6, p.118) specifies the operations of the UCM sub-map in Figure 6.3, in Chapter 6. The sub-map models all the activities related to sending requests over the network. The class encompasses a set of three activities each describing a complete scenario (as perceived within the sub-map) to handle each type of request.

For example, **Activity #1** specifies, with an Object-Z operation schema expression, the scenario to forward a request to check an invoice. It includes the sequence of operations, together with some conditions, in square brackets, that need to be fulfilled.

If the system was limited to this sub-map, then, the specification would readily include a complete description of scenarios for the entire system.

Example 2: As mentioned above, in the discussion about the traceability of the UCM-OZ specification, the class schema *ClsHelper* (see Chapter 6, p.134) also includes three sets of operations, that, each specifies, a sequence of activities that may be performed by the system when acting as *Helper*.

For example, the operation *startFromS*1 specifies the scenario to return an item ($g4$), from the beginning, until the returned item is shipped to its provider.

It appears that the scenario specification seems to be naturally encompassed by the UCM-OZ specification. It is important to notice that a scenario description is included in a UCM-OZ class, whenever an attempt is made to specify a UCM component, which includes other UCM elements.

Next the leftward validation phase is discussed.

8.4.3 The leftward validation

This validation phase aims to discuss the ability of the UCM-OZ specification, relative to the following properties: *creativity* ($p7$), *networking* ($p10$) and *Integrity* ($p11$).

Creativity($p7$)

As mentioned earlier (in Section 8.3.3, p.177), the reason for the creativity property, is to evaluate the ability of the specification technique or process, to stimulate the specifier's imagination about the required functionalities of the system.

It is important to observe how the UCM-OZ specification is structured into groups of classes, in which the following remarks are relevant:

- Each group aims to specify a specific aspect of the system. Below are two examples of classes:

Example 1: The classes *ClsMainStartPoints*, *ClsInitChecking* (Section 6.3.5, p.132), and *ClsHelper* (see Section 6.3.5, p.134) together, aim to specify the reaction of the system to an

external event, such as *a customer wanting to return an item or to pay a credit*; and also to prepare the requests that need to be sent to other agencies via the network.

Example 2: In Section 6.3.2, the classes: *ClsTimer* (on page 115), *ClsInterface* (on page 116), *ClsNetInterface* (on page 116), *ClsNetComTemp* (on page 117), and *ClsRequest* (on page 118) when taken together, aim to specify the behaviour of the system with regard to the submission of requests via the network, implementing a mechanism to control network failure and maintaining a list of unresolved requests, until they are successfully transmitted over the network.

- Some classes in a group specify individual operations in detail, whereas one of them specifies how those individual operations are (architecturally) structured within the system, hence presenting an operational view of the system being specified.

  In Example 1 above, the class *ClsMainStartPoints*, for example, specifies the operations $s1$, $s2$, and $s3$ to initialise, respectively, the scenario to return an item, to replace an item, and to pay a credit. These operations are activated to respond to an external event initiated by a customer. The origin of these operations is clearly justified. They result from the Object-Z transformation of the three start-points $S1$, $S2$, and $S3$ of the UCM model, in Figure 6.2. Many other specified operations have a similar origin. For example, all the operations within the class *ClsTimer*, to control the network failure and recovery, are due to the UCM concept of Timer and the associated pre-defined mechanism to control network failure (see Buhr and Casselman [20]).

  The class *ClsRequest*, in Example 2, includes, for example, the expression:

  Activities #3
  $[id3? : Identifier, cust? : Customer, amnt? : Money, ad? : Address] \bullet s3 \fatsemi reqUpdateAccount \fatsemi e1$

  This expression, labeled `Activities #3` shows how the three operations $s3$, *reqUpdateAccount* and $e1$ are composed in sequence, to get a request to update a customer's account updated, after a credit has been paid. It also indicates, in the square bracket, the condition for the process to be started.

- Groups are inter-dependent. For example, after a request is prepared as specified by the group in Example 1, it has to be transmitted via the network, hence involving the classes in Example 2. This specific aspect of inter-dependence between classes in Example 1, and those in Example 2, is specified by the class *ClsHelper* that inherits the class *ClsRequest*.

The next section discusses the ability of the UCM-OZ to describe issues related to network communication between agencies.

Networking ($p10$)

The context and purpose of the networking property was discussed in Section 8.3.3 (on page 178). Because of the inherent nature of this application, network communication is a pre-requisite to enable transactions between agencies.

The above discussion on the creativity property, shows in Example 2, that a set of five class schemas are necessary in the UCM-OZ specification, to specify operations to control network communication, and describe a mechanism for network failure and recovery. It may be argued that the question of including certain functionalities in a specification is subjective because, different ways of solving the same problem are possible and the choice is left to the specifier. However, the process of transforming the UCM (Figure 6.2 in Chapter 6) model of the case study, into Object-Z, clearly shows that the specified functionalities and the structuring of the UCM-OZ specification, were primarily driven by UCM concepts, such as the UCM Timer and Components.

The following section aims to analyse the integrity of the operations specified in the UCM-OZ specification within the application domain.

Integrity ($p11$)

The discussion about the creativity property of the UCM-OZ specification revealed that the specification can be structured with a set of inter-dependent groups of classes. Each such group of classes, specifies a specific task expected from the system. More importantly, within a group, at least one of the class schemas clearly describes how individually specified operations are composed to perform the task. It is similarly argued that a specification already encompasses the concept of scenario from the UCM technique which is inherently scenario-based. Owing to its scenario-based nature, re-constructing complete scenarios from a specification, derived from a UCM, is relatively easy, because some classes already include composite operations to specify UCM scenarios.

Such a composition of operations, representing a (segment of a) scenario in progress, depicts the operational view of the system and hence, shows how each function within the specification, is integrated into the system as a whole. It has also been shown that having the ability of the specification to provide scenario specification, may facilitate the operational

feasibility study of the system.

Next the rightward validation phase, that addresses the qualities of the specification related to the notation language and associated tool support, is presented.

8.4.4 The rightward validation

This phase is where various tools associated with the specification notation language, may be applied to the specification under consideration. It is also the moment during which the applicability of such tools may be addressed. In this dissertation, the aim is not to apply Z and Object-Z tools to the UCM-OZ specification, but to study the ability of the specification to facilitate their use.

Correctness ($p12$)

The discussion of the correctness of the UCM-OZ specification, in the context of this dissertation, is limited to the syntax and semantic correctness. The intent is not to practically apply Object-Z tool support (such as type checkers and semantic analysers), to the specification, but to analyse their applicability. Thus, reference is made to the discussion in Section 8.3.4, where a similar analysis, about the correctness of the Z-OZ specification, was performed. The reason why further discussion about the applicability of the Object-Z tool support does not provide more insight, is that each of the two specifications (Z-OZ and UCM-OZ), resulted from the application of the same framework to a Z document, and the same Latex package (OZ.sty) for encoding the Z and Object-Z documents.

Next, the Internal consistency property is discussed.

Internal consistency ($p13$)

Similar to the discussion in Section 8.3.4 (on page 180), a twofold manual check is conducted to investigate the internal consistency of the UCM-OZ specification. Firstly, contradictions within each individual class schema are checked. Then, the analysis is extended to the class level, where the consistency between inter-related classes, is investigated. The result of such an investigation is the following:

1. The pre-condition ($time?\,0$) of the operation *setup*, within the class *ClsTimer*, is syntactically incorrect. This can be easily corrected, however if not detected, its interpretation can be misleading. The correct version is $time? > 0$. For example, $time?\ 0$ may be interpreted as a function, to which the argument 0 is applied.

2. The class *ClsLocalSales* specifies an interface to query the status of a local sales system of a company, without allowing direct access to its components. Hence, it may not be necessary to include the *INIT* operation, as the system under consideration has no control over any local sales system that normally exist prior to the system being specified.

3. Consider the following two operations from the class *ClsCheckPoint*:

$$IN1 \mathrel{\widehat=} [id? : Identifier;\ inv? : Invoice] \bullet plugin4Inv.s1 \tag{8.24}$$

$$IN2 \mathrel{\widehat=} [id? : Identifier;\ cust? : Customer] \bullet plugin4Cust.s2 \tag{8.25}$$

where *plugin4Inv* is a pointer to an object of the class *ClsCheckInvoice*, and *plugin4Cust*, a pointer to an object of the class *CheckCustomer*. The expression of the operation $IN1$, described with formula 8.24, specifies the fact that the system selects one of the incoming requests, temporarily keeps it in the variable *collectedReq*, and submits it to the object *plugin4Inv* for checking. The variables inside the square brackets are used to indicate the values of the selected request. The problem here, is that the format of a selected object (*identifier*, *invoice*), does not match that of an incoming request (*identifier*, *request*) and the additional function is presented to show, for example, how the information on the invoice is generated from a request. A similar observation can be made with the operation $IN2$, in the Formula 8.25.

The above problem may however, result from a simple omission or negligence because the class *ClsGlobalVariables*, inherited by the class *ClsCheckPoint*, readily defines the functions:

$$invoiceInReq : Request \pfun Invoice$$
$$customerInReq : Request \pfun Customer$$

with the function *invoiceInReq* mapping each request to the invoice, for which the request was created, and the function *customerInReq* mapping each request to the customer, for whom the request was created.

The next section aims to investigate the specification, in order to reveal properties that are irrelevant to the problem in the case study and the problem solution.

Minimality ($p14$)

With large systems, it may be hard to decide on the minimality of a software specification since one has to check the entire specification to identify redundant and irrelevant components. To analyse the UCM-OZ specification, a twofold strategy is adopted: first, the specification is investigated to identify irrelevant class schemas within the system. Secondly,

each class schema is investigated to identify irrelevant components within the class.

The traceability matrix in Figure 8.9, relating class schemas to the specified UCM elements, reveals that each class is relevant, since every class contributes to specifying at least one UCM element that helps model the system.

A detail analysis of classes does not reveal any unused component or variable. However, it remains difficult to definitive on this issue because not all operations are completely specified in the specification. Examples are the operation *respond* in the class *ClsCheckPoint*, and the operations *evalItem* and *updateCustAccount* in the class *ClsUpdatePoint*.

Next the maintainability property is discussed.

Maintainability ($p15$)

It is argued in this dissertation that the ease of modifying a software specification depends on its structure, style and the ease of reading and understanding the specification. The readability and understandability of the UCM-OZ specification were discussed earlier in Section 8.4.2. The structuring of the specification was also discussed. For example, in Section 8.4.3, (on page 193), it was demonstrated that the specification is structured in such a way that it forms inter-dependent groups of classes, where each group performed a specific operation in the system. It is important to notice how the structuring of the specification, is based on the architectural structure of the UCM model from which the specification was derived. With the support of the traceability matrices in figures 8.8 and 8.9, it becomes less difficult to trace the part of the UCM model and hence, the class or group of classes that may be affected when there is a change at the User requirement level.

The next section discusses the downward validation phase.

8.4.5 The downward validation

During the downward validation phase, the UCM-OZ specification is analysed to evaluate its ability to facilitate the technical feasibility of the system and its ability to be executed.

Technical feasibility

The issues related to the ability of the Z-OZ specification to facilitate the technical feasibility of the system under construction were addressed in Section 8.3.5 (on page 183). The discussion conducted in that section remains credible in the case of the UCM-OZ specification,

since the discussion is more about the refinement of Object-Z specifications. However, the impact of the UCM in such a study should not be omitted.

Originally, the UCM technique is intended to represent the static and dynamic behaviour of a software system and facilitate the architectural projection of such a system at the requirements level (Buhr [18], Buhr and Casselman [20], Buhr et al. [21]). Hence, having a complete picture (including the static and dynamic behaviour, as well as the architecture) of an envisioned software product may be valuable for a feasibility study of the system. One of the advantages of the UCM-OZ specification is the fact that the specification inherited some of those qualities of the UCM technique. For example, as discussed in Section 8.4.3, the architectural structuring of the UCM model is reflected in the UCM-OZ specification. UCM scenario models (represented with path elements) are also specified in certain classes (e.g. *ClsHelper*), including the sequential composition of operations.

Executability

The discussion about the executability of the Z-OZ specification in Section 8.3.5 is reconsidered here since it is focused mainly on the Object-Z and associated tool support. An important advantage of the UCM-OZ specification, is the support it may have from the UCM model (on which it is based), in relation to representing and understanding the behaviour of the envisioned system and its architecture.

8.5 Chapter summary

This Chapter presented the list of stakeholder expectations and user requirements described in the case study, that was developed in Chapter 3; the quality of a specification for the case study is judged relative to them. A sample list of properties that are expected from a satisfactory specification, and a set of criteria to evaluate each of the two proposed specifications, were identified hence, defining the scope and limits for the validation process. The framework proposed in Chapter 7 was applied to each of the specifications Z-OZ and UCM-OZ specifications, respectively, in sections 8.3 and 8.4. Only one iteration was considered because the aim was to capture the actual status of each of the specifications.

This chapter has therefore, provided a mechanism to evaluate the quality of a specification relative to a given set of goals and user requirements and demonstrated the applicability of the proposed specification validation framework.

The analysis of the validation results obtained in this chapter is the focus of the following chapter.

Chapter 9

Analysis

A detailed discussion of some selected properties of the Z-OZ and UCM-OZ specifications of the case study was presented in Chapter 8. This chapter analyses the results of those findings, aiming to compare the two specifications. To this end, the approach used to guide the comparison is first presented.

9.1 Analysis approach

The comparison strategy is depicted in Figure 9.1 which summarises all that was done in the preceding chapters, and illustrates the comparison process. The two specifications Z-OZ

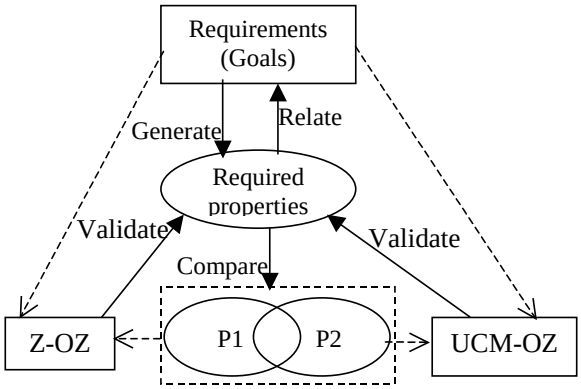


Figure 9.1: Basic comparison strategy

and UCM-OZ of the case study were constructed based on the initial goals. These were validated in Chapter 8, where a sample set of properties were generated and related to the goals. This chapter aims to compare the two specifications relative to the selected properties, by determining amongst them those that are fulfilled by each of the specifications. To do so,

guidelines are needed that may set up the boundaries and context of the comparison giving an idea on how to proceed. This section states such guidelines in the following paragraphs:

Guideline #1 : The analysis is based on the discussions about the qualities of the Z-OZ and UCM-OZ specification presented, respectively, in sections 8.3 and 8.4. The specification construction processes are illustrated in Figure 8.1 (for the Z-OZ) and Figure 8.2 (for the UCM-OZ) are also considered.

Guideline #2 : For each property p, the analysis is to identify which of Z-OZ and UCM-OZ specifications adhered most to the property. To this end, the previous discussions (performed in Chapter 8) about each of the specifications are studied, as well as their construction processes.

Guideline #3 : In the context of this analysis, to enable the comparison between the two specifications, each property is evaluated to a boolean value 1 or 0 (true or false). For example, when a property p is found to be more encompassed by one of the two specifications, p is assigned the value 1 (or true) for the specification in which it is more present and the value 0 (or false) for the other specification.

Guideline #4 : For a property p, if there are not enough arguments to justify in which specification the property is more embraced, p is assumed to be present in both specifications (since we are more interested in when there is a difference).

The following discussion uses the above guidelines to analyse and compare the two specifications, Z-OZ and UCM-OZ of the case study.

9.2 Comparing Z-OZ and UCM-OZ specifications

Table 9.1 summarises the analysis of the specifications aiming to compare the availability of properties in those specifications. In line with Guideline #3, each property is assigned a boolean value.

9.2.1 Table of comparison

In the light of the above guidelines (Guideline #1 to Guideline #4), Table 9.1 is generated. The specifications are represented in columns, and properties in rows. Each cell contains a boolean value (true or false), indicating whether the specification in the column encompasses the property or not.

Properties		Z-OZ	UCM-OZ
readability	$p1$	true	true
understandability	$p2$	false	true
clarity	$p3$	false	true
operational feasibility	$p4$	false	true
technical feasibility	$p5$	true	true
executability	$p6$	true	true
creativity	$p7$	false	true
traceability	$p8$	false	true
completeness	$p9$	true	true
networking	$p10$	false	true
integrity	$p11$	true	true
correctness	$p12$	false	true
internal consistency	$p13$	false	true
minimality	$p14$	true	false
maintainability	$p15$	false	true
modifiable	$p16$	false	true

Table 9.1: Table of comparison

When considering the two discussions about the nested inheritance of the Z-OZ specification (based on Table 8.2) and that of the UCM-OZ (based on Table 8.7) from Chapter 8, it appears that UCM-OZ presents some remarkable advantages.

- Percentages wise, UCM-OZ has fewer classes from level 2 to level 4 than Z-OZ. At level 1, the representation is the same, at level 2, the percentage of the UCM-OZ is slightly lower, but at levels 3 and 4, it decreases.

- Although a specification having classes at the higher levels (levels 4 and 5) is seen as disadvantageous, in the case of the UCM-OZ, an analysis of the (internal) structure and contents of those classes, reveals an advantage. As demonstrated in the previous chapter, the classes: *ClsRequest*, *ClsBeneficiary*, and *ClsHelper*, do not primarily aim to specify other functionalities of the system but, rather to reconstruct the sequence of operations specified in other classes, to form a complete scenario, or part thereof.

- The above mentioned classes also have the advantage of revealing the inter-communication between sub-systems, since they attempt to specify, through scenario reconstruction,

a UCM path segment within, a UCM team component in which the path traverses multiple contained UCM elements, on which other classes are based.

For example, the expression *startFromS2* in the class *ClsHelper*, specifies the path segment included in the team component *Helper*, that begins at the UCM start-point *S2*, passes across the team component *InitChecking* and the stub element *NetControl*, exits the component *Helper*, then re-enters and passes through the team component *Transit_point*, (within which the UCM object *Store* is traversed), before the path segment finally leaves the component *Helper*.

The above advantages contributed to making some of the properties being evaluated to "true", for the UCM-OZ specification, and consequently to "false", for the Z-OZ one. The properties directly affected are: the *understandability*, *clarity* and *traceability*. Additionally, no clear justification is found to show that either of the specifications is more *readable* than the other. A potential advantage of UCM-OZ, may be that most of the classes in the specification, are of a reasonable size (do not exceed an A4 page). But, this argument is not being sufficiently convincing because Z-OZ has a very small number of classes, but only one of which (class (*ClsSystem*)) is enormous. Considering the average size, the real difference in class sizes in both specifications is therefore not easy to observe. The discussion (in Chapter 8) about the *readability* property in both cases, mainly focuses on the notation of Object-Z.

Other properties for which neither of the two specifications ought to make a significant difference are: *technical feasibility*, *executability*, and *completenes*.

(a) With regard to *technical feasibility*, the analysis in Chapter 8, focuses on demonstrating that an Object-Z specification, in general, can be refined into a software product. Some cases of such transformation techniques were briefly presented. It was also suggested that the presence of certain properties in a specification along with a good structuring of components therein, will facilitate a technical feasibility study. As no example was presented to show, for example, the contribution of the suggested properties in refining a specification with a specific method, it remains hard to decide which of the two specifications, Z-OZ and UCM-OZ, best facilitate the technical feasibility of the system.

(b) Regarding the *executability* property, it is hard to objectively decide which of the two specifications would be easier to be executed. Since the discussion from Chapter 8 focuses on the general case of animating Object-Z specifications and associated tools. This case is similar to that of the *technical feasibility* and can, therefore, be analysed in the same vein. Because they both concern the application of some techniques or methods to the specifications: the refinement techniques for the *technical feasibility* and animation techniques for the *executability*.

(c) With regard to the *completess*, the discussion in Chapter 8 does not go beyond the three user-requirements, $g4$, $g5$, and $g6$. It attempts to demonstrate that each of the two specifications fully addresses each of those requirements. Hence, it remains difficult to say which one of the two specifications is more comprehensive.

The very small number of classes in the Z-OZ specification and its completeness make the Z-OZ specification minimal than the UCM-OZ with more classes. But an important aspect to consider is that the Z-OZ specification specifies operations of the system at a high level of abstraction close to the user requirements level. Owing to the use of the UCM technique at the user requirements level, with UCM-OZ, operations are first refined into the UCM responsibility points, and grouped within abstract components before being formalised. Such an analysis phase provided the specification with the advantage of being more creative (*creativity* property evaluated to "true"). The efforts to specify the *UCM* components (that are well known entities in the UCM, e.g Network, Timer, etc.) provided the specification not only with the creativity ability, but also enable the specification to explicitly describe issues related to network communication (*networking* property).

Further advantages of using the UCM technique is that the grouping of operations within UCM components, limits the application of the Z technique, to specific components of the system, and introduces, at an early stage of the design, architectural structuring of system components into the final specification. For example, most of the Object-Z classes, within the UCM-OZ specification, are obtained and interrelated from UCMs components, and the inter-Communication between them.

In the light of the above analysis, the next section uses the data in Table 9.1, to attempt to determine how well each of the two specifications satisfies a stakeholder's expectations.

9.2.2 Satisfying goals and expectations

The Z-OZ specification contains a very few classes. This may suggest that the specification was constructed at a very high level of abstraction and hence, although it may be readable and clear, it still remains difficult to grasp, as a significant level of detail is not demonstrated as shown above with the technical feasibility.

To further show how the properties support the initial goals, and users requirements, two instances of the set S of formulae (see Section 8.2.1) developed in Chapter 8 are presented. The value assigned to each property is taken from Table 9.1.

Z-OZ

$$
\begin{array}{l}
\{ \\
\quad \mathrm{true} + \mathrm{false} + \mathrm{false} \rightsquigarrow g1, \\
\quad (\mathrm{true} + \mathrm{false} + \mathrm{false}) + (\mathrm{false} + \mathrm{true} + \mathrm{true}) \rightsquigarrow g2, \\
\quad \mathrm{false} + \mathrm{false} + \mathrm{false} + \mathrm{false} + \mathrm{true} \rightsquigarrow g4 \wedge g5 \wedge g6, \\
\quad (\mathrm{false} + \mathrm{true} + \mathrm{true}) + (\mathrm{false} + \mathrm{false} + \mathrm{true}) \rightsquigarrow g3, \\
\quad \mathrm{false} + \mathrm{false} + \mathrm{true} + \mathrm{false} + \mathrm{false} \\
\}
\end{array}
$$

UCM-OZ

$$
\begin{array}{l}
\{ \\
\quad \mathrm{true} + \mathrm{true} + \mathrm{true} \rightsquigarrow g1, \\
\quad (\mathrm{true} + \mathrm{true} + \mathrm{true}) + (\mathrm{true} + \mathrm{true} + \mathrm{true}) \rightsquigarrow g2, \\
\quad \mathrm{true} + \mathrm{true} + \mathrm{true} + \mathrm{true} + \mathrm{true} \rightsquigarrow g4 \wedge g5 \wedge g6, \\
\quad (\mathrm{true} + \mathrm{true} + \mathrm{true}) + (\mathrm{true} + \mathrm{true} + \mathrm{true}) \rightsquigarrow g3, \\
\quad \mathrm{true} + \mathrm{true} + \mathrm{false} + \mathrm{true} + \mathrm{true} \\
\}
\end{array}
$$

Further analysis of S is beyond the scope of this dissertation. However, it may be noticed that the more the properties on the left side of a formula are evaluated to true, the more the goals on the right side tend to be supported by the specification.

9.3 Chapter summary

This chapter exploited the discussion from Chapter 8, to highlight amongst the sample properties used for this study, those on which the use of UCMs may have brought improvements. The approach for the comparison suggested some sample guidelines and a table partitioning the properties, according to whether they are satisfied by a specification, or not. An important advantage of this chapter is the attempt to use formulae to relate expected properties to stakeholders' expectations and users, requirements, hence enlightening a rich area for future research, that may involve investigating the feasibility of using a reasoning mechanism to discharge proof obligations surrounding supportness aspects. E.g., using the new Prover 9 reasoner McCune [58]to establish levels of support of the system goals defined as part of the specification.

The final chapter concludes the dissertation, and presents directions for future work in this area.

Chapter 10

Conclusion and Future work

This chapter concludes this dissertation. It first presents the main findings and relates them to the research problem investigated in this work. In a short discussion, it enlightens the extent to which the research questions, posed in Chapter 1, were addressed. Thereafter, the advantages and possible limitations of the work are presented. The chapter ends by considering future potential research areas to complement the findings.

10.1 Research questions and the main findings

This dissertation evaluated the impact of using UCM techniques at the requirements elicitation phase, during the construction of a Z and Object-Z specification. The aim being to improve the construction process of Z and Object-Z, by providing a methodology whereby a step-by-step mechanism along with construction guidelines, are provided. To this end, the first research question posed was the following:

RQ1: *Are UCM models transformable to Z and Object-Z specifications? In other words, can UCM models of a system be used as inputs for Z and Object-Z specifications?*

Chapter 5 proposed a generic framework whereby a UCM model of a system may be transformed into Z and Object-Z specifications. The framework was used in Chapter 6 to translate the UCM model of the case study, into Z and Object-Z specifications. The framework was compiled and presented as a research paper at the 7th International Workshop on Modelling, Simulation, Verification and Validation of Enterprise Information Systems (MSVVEIS 2009)[23].

Demonstrating the transformability of UCMs, ensures that the technique may be employed at an early stage of Z and Object-Z specification construction. However, this does not guarantee any improvement in the quality of the final specification. Hence, the next research

question posed:

RQ2: *What would be the impact of UCMs on the quality of a Z and Object-Z specification obtained by transforming a UCM model?*

In Chapter 8, each of the two Object-Z specifications Z-OZ, (constructed without using any UCM), and UCM-OZ, (derived from the UCM model of the case study), was validated. A sample set of properties expected from a suitable specification was considered and discussed individually for each specification. The analysis of the two specifications relative to those properties, conducted in Chapter 9 (see Table 9.1), shows that:

- Eleven (11) properties out of sixteen (16) (representing 69.75%) were actually improved on in the UCM-OZ specification.
- One (1) property out of sixteen (representing 6.25%) was negatively affected. The Z-OZ specification contains fewer of specification elements than its counterpart UCM-OZ.
- Four (4) property out of sixteen (representing 25%) do not reveal any change as a result of using use of UCM.

The main ideas in chapters 7 and 8 that were used to support our specification validation approach were developed as a research paper and presented at the SAICSIT 2010 Conference [24]. Another research paper was synthesised from chapters 8 and 9[25]. The paper developed the basic strategy for comparing two software specifications based on the validation approach in chapters 7 and 8.

The last concern in this dissertation investigated the influence that a UCM may have in the process of constructing a Z, and an Object-Z, specification. Thus the last research Question posed:

RQ3: *What would be the impact of UCMs in the process of constructing Z and Object-Z specifications? The point is for example, to ensure that the use of a UCM or any other semi-formal technique facilitates the building of Z and Object-Z specifications.*

In Chapter 3, an attempt to generate a Z specification directly from a set of user requirements, led to the following observations:

- The resulting Z and Object-Z specifications were at the same level of abstraction as the user requirements, specifying the user-view of the system. This observation suggests the necessity of deriving the system requirements, before formalising them, if the specification of the system-view is required.

- The specification was progressively built-up in different phases, during which building blocks were provided for each and subsequent phases, to construct other specification components. E.g. Z basic types identified in the requirements, were used to build state schemas, which are needed to specify operation schemas. Thus, modifying a single element in a Z and an Object-Z specification, may affect many other elements at different levels in the specification. This observation suggests that with large systems, it may not be adequate to start formalising (user requirements) at the requirements elicitation and analysis stage, since at that stage, requirements are, in general, subject to many subsequent changes.

The above observations clearly suggest the need for a more flexible and user-friendly technique at the initial stages of requirements capturing and analysis, to complement Z and Object-Z. As mentioned in Chapter 1, this suggestion is strongly supported in the literature (see for example Abrial [2], Sommerville [82], van Lamsweerde [95]). Observations made, during the construction of the UCM model of the case study, in Chapter 3 and listed in Section 3.3.5, suggests UCM as a suitable candidate.

The framework for transforming UCMs to Z and Object-Z (in Chapter 5), and its application to the case study (in Chapter 6) shows that, in the process of generating Z and Object-Z specifications of a system, UCMs readily structure the system into a set of interrelated components. Hence, allowing Z to be applied to individual components, alleviate the difficulty of applying the formal technique to large projects, since each component is of a reduced size and represents a manageable sub-system. In the same vein, UCM components readily provide for Object-Z meta-classes, that may later be completed with Z schemas, obtained by formalising the components from which meta-classes are generated. Some of the important advantages of this work are summarised in the next section.

10.1.1 Advantages

This dissertation suggests a step-by-step construction process for Z and Object-Z specifications. It uses the Use Case Map notation technique as an intermediate step, to capture and structure requirements from users, to prepare the system components to be formalised, and to produce meta-classes for Object-Z. Hence, this resolves the problem of people being reluctant to use formal techniques (as noted by Abrial [2, page 766] below):

People are quite reluctant to use such methods mostly because it necessitates to modify the development process in a significant fashion. As it is well known, such development processes are hard to develop and even harder to put in place so that the working engineers are using

them.
By putting in place a framework, this work has contributed to improve the constructivity of Z and Object-Z specifications and provided additional guidance. It has equally helped to render formal reasoning suitable for designing software on an incremental basis (Larman and Basili [50]). Since UCM models a system as a whole, and structures it into inter-connected sub-systems that may then, be developed separately, and later integrated progressively when the required components are built.

The proposed specification process also suggests using the Z notation to interact with a UCM, aiming to detect errors in the UCM documents during the requirements capturing and modelling phase (see Fig. 5.1), for example, by calculating the pre-conditions for each UCM responsibility point. This suggestion contributes to resolve the problem of applying formal techniques to requirements documents, with potentially adverse errors, that may only be discovered later in the process during global testing (Abrial [2]).

As mentioned before, the process may also contribute to improve the quality of the final specification, by reinforcing about 70% of all the properties required for a suitable specification (see Chapter 9).

The UCM transformation framework in Chapter 5, lays the foundation for further research that may provide an interactive graphical user interface environment for building Z and Object-Z specifications based on UCMs.

Chapter 7 established a framework which embodies a 4-way validation strategy to iteratively, develop and measure a software specification against user-requirements, the application domain, the notation language, and finally the envisioned operational system, that may be obtained through appropriate refinement.

Another important advantage of this work is that it provides guidelines to formalise the method of applying existing techniques to help evaluate, and compare and hence, enable the selection, of an appropriate specification, from a number of possible alternatives.

10.2 Future work

Some of the important future research for further investigation that emanates from this dissertation includes:

- Automating the generation of Z and Object-Z specifications, by providing, for in-

stance, an interactive interactive graphical-user-interface environment for building Z and Object-Z specifications based on UCMs.

- Some empirical work needs to be carried out in industry to help strengthen and refine the specification validation framework, proposed in Chapter 7. Such work may aim to evaluate the applicability of the framework to large projects, and other types of specifications (e.g. UML, URN, Petri Nets, etc.).

- Further research is also needed to refine each of the four validation phases of the framework in Chapter 7, and eventually to automate the validation process.

- The attempt, in chapters 8 and 9, to use mathematical expressions to relate specification properties, to system goals, and user requirements, opens up a rich area for future research that may consist of investigating the feasibility of using a reasoning mechanism, (for example the new Prover 9 Reasoner, McCune [58]), to help establish levels of support for intended goals and system requirements.

- Each of the UCM, Z and Object-Z notations, investigated in this dissertation, only describes functional requirements. Therefore, an interesting and challenging research area would be to explore the likelihood of enhancing non-functional requirements in the process of constructing Z and Object-Z from a Use Case Map.

Bibliography

[1] ITU-T, Recommendation Z.151 (11/08), User Requirements Notation (URN)-Language definition., November 2008. URL http://www.itu.int/rec/T-REC-Z.151/en.

[2] Jean-Raymond Abrial. Formal methods in industry: achievements, problems, future. In *Proceedings of the 28th international conference on Software engineering*, ICSE '06, pages 761–768, New York, NY, USA, 2006. ACM. ISBN 1-59593-375-1. doi: http://doi.acm.org/10.1145/1134285.1134406. URL http://doi.acm.org/10.1145/1134285.1134406.

[3] Edward B. Allen. Typesetting Technical Reports that Include Z Specifications Using LaTeX, 2006.

[4] Daniel Amyot. Use Case Maps: Quick Tutorial, September 1999.

[5] Daniel Amyot. Introduction to the user requirements notation: learning by example. *Computer Networks*, 42(3):285–301, June 2003. ISSN 1389-1286. doi: 10.1016/S1389-1286(03)00244-5. URL http://dx.doi.org/10.1016/S1389-1286(03)00244-5.

[6] Daniel Amyot. Use Case Maps as a Feature Description Notation. In *FIREworks Feature Constructs Workshop*, pages 27–44. Springer-Verlag, May, 2000.

[7] Daniel Amyot and Gunter Mussbacher. URN: Toward a New Standard for the Visual Description of Requirements. *E.Sherratt (Ed.) SAM 2002*, pages 21–37, 2002.

[8] Daniel Amyot and Gunter Mussbacher. User requirements notation: The first ten years, the next ten years (invited paper). *Journal of Software*, 6(5), 2011. URL http://ojs.academypublisher.com/index.php/jsw/article/view/0605747768.

[9] Daniel Amyot, Xiangyang He, Yong He, and Dae Yong Cho. Generating Scenarios from Use Case Map Specifications. In *QSIC '03: Proceedings of the Third International Conference on Quality Software*, page 108, Washington, DC, USA, 2003. IEEE Computer Society. ISBN 0-7695-2015-4.

[10] Robert Balzer and Neil Goldman. Principles of good software specification and their implications for specification languages. In *AFIPS '81: Proceedings of the May 4-7, 1981, national computer conference*, pages 393–400, New York, NY, USA, 1981. ACM. doi: http://doi.acm.org/10.1145/1500412.1500468.

[11] Grady Booch, James Rumbaugh, and Ivar Jacobson. *The Unified Modeling Language User Guide*. Addison-Weseley, 2nd edition, 2005. ISBN 0-321-26797-4.

[12] F. Bordeleau and R. J. A. Buhr. The UCM-ROOM Design Method: from Use Case Maps to Communicating State Machines. In *Conference on the Engineering of Computer-Based System*, Monterey, USA, 1997.

[13] Jonathan Bowen. *Formal Specification and Documentation Using Z: A Case Study Approach*. International Thomson Computer Press, London / Boston, 1996. ISBN 1-85032-230-9.

[14] Jonathan P. Bowen and Michael G. Hinchey. Ten Commandments of Formal Methods ...Ten Years Later. *Computer*, 39:40–48, January 2006. ISSN 0018-9162. doi: 10.1109/MC.2006.35. URL `http://portal.acm.org/citation.cfm?id=1110638.1110672`.

[15] Frederick P. Brooks. No Silver Bullet: Essence and Accidents of Software Engineering. *Computer*, 20(4):10–19, 1987. ISSN 0018-9162. doi: http://dx.doi.org/10.1109/MC.1987.1663532.

[16] Paul C. Brown. *Implementing SOA: Total Architecture in Practice*. Addison-Wesley, 1st edition, 2008.

[17] R. J. A. Buhr. Use Case Maps: A New Model to Bridge the Gap Between Requirements and Design. *SCE 95- Conttribution to the OOPSLA 95 Use Case Map Workshop*, pages 1–4, 1995.

[18] R. J. A. Buhr. Understanding Macroscopic Behavior Patterns with Use-Case Maps. In Mohamed E. Fayad, Douglas C. Schmidt, and Ralph E. Johnson, editors, *Building Application Frameworks - Object-Oriented Foundations of Framework Design*, pages 415–439. John Wiley & Sons, New York, 1999.

[19] R. J. A. Buhr. Making Behaviour a Concrete Architectural Concept. In *HICSS'99, 32nd Annual Hawaii International Conference on System Sciences*, Hawaii, USA, 1999.

[20] R. J. A. Buhr and R. S. Casselman. *Use Case Maps for Object-Oriented Systems*. Printice Hall, USA, 1995.

[21] R. J. A. Buhr, D. Amyot, M. Elammari, D. Quesnel, T. Gray, and S. Mankovski. Features-Interaction Visualiation and Resolution in an Agent Environment. In K. Kimbbler and L. G. Bouma, editors, *FIW'98, Fifth International Workshop on Feature Interaction in Telecommunications and Software Systems*, Lund, Sweden, 1998. IOS Press, 135-149.

[22] David Carrington and Graeme Smith. Extending Z for Object-Oriented Specifications. 5th Australian Software Engineering Conference, (Sydney), May, 1990.

[23] Cyrille Dongmo and John A. van der Poll. Use Case Maps as an Aid in the Construction of a Formal Specification. In Daniel Moldt, Juan Carlos Augusto, and Ulrich Ultes-Nitsche, editors, *MSVVEIS*, pages 3–13. INSTICC PRESS, 2009. ISBN 978-989-8111-90-6.

[24] Cyrille Dongmo and John A. van der Poll. A Four-Way Framework for Validating a Specification. In Paula Kotze, Aurona Gerber, Alta van der Merwe, and Nicola Bidwell, editors, *SAICSIT*, pages 46–59. ACM PRESS, 2010. ISBN 978-1-60558-950-3.

[25] Cyrille Dongmo and John A. van der Poll. Evaluating software specifications by comparison. In *Proceedings of the South African Institute of Computer Scientists and Information Technologists Conference on Knowledge, Innovation and Leadership in a Diverse, Multidisciplinary Environment*, SAICSIT '11, pages 87–96, New York, NY, USA, 2011. ACM. ISBN 978-1-4503-0878-6. doi: http://doi.acm.org/10.1145/2072221.2072232. URL `http://doi.acm.org/10.1145/2072221.2072232`.

[26] Roger Duke and Gordon Rose. *Formal Object-Oriented Specification Using Object-Z*. Macmillan, Basingstoke, 2000. ISBN 0333801237.

[27] Roger Duke, Paul King, Gordon A. Rose, and Graeme Smith. The Object-Z Specification Language. In Timothy D. Korson, Vijay Vashnavi, and Bertrand Meyer, editors, *TOOLS (5)*, pages 465–484. Prentice Hall, 1991. ISBN 0-13-923178-1.

[28] Steve Dunne. Understanding Object-Z Operations as Generalised Substitutions. In Eerke A. Boiten, John Derrick, and Graeme Smith, editors, *IFM*, volume 2999 of *Lecture Notes in Computer Science*, pages 328–342. Springer, 2004. ISBN 3-540-21377-5.

[29] Sophie Dupuy-Chessa and Lydie du Bousquet. Validation of UML Models Thanks to Z and Lustre. In *FME '01: Proceedings of the International Symposium of Formal*

Methods Europe on Formal Methods for Increasing Software Productivity, pages 242–258, London, UK, 2001. Springer-Verlag. ISBN 3-540-41791-5.

[30] Evans. *Domain-Driven Design: Tacking Complexity In the Heart of Software*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 2003. ISBN 0321125215.

[31] Wernher R. Friedrich and John A. van der Poll. Towards a Methodology to Elicit Tacit Domain Knowledge from Users. *Interdisciplinary Journal of Information, Knowledge, and Management*, 2:119–193, 2007.

[32] Norbert E. Fuchs. Specifications are (preferably) Executable. *Softw. Eng. J.*, 7:323–334, September 1992. ISSN 0268-6961. doi: http://dx.doi.org/10.1049/sej.1992.0033. URL `http://dx.doi.org/10.1049/sej.1992.0033`.

[33] Martin Gogolla and Mark Richters. On Combining Semi-Formal and Formal Object Specification Techniques. In *Recent trends in algebraic development techniques: 12th international workshop, WADT97*, pages 238–252. Springer, 1998.

[34] Andrew M. Gravell. What is a Good Formal Specification? In *Proceedings of the Fifth Annual Z User Meeting on Z User Workshop*, pages 137–150, London, UK, 1991. Springer-Verlag. ISBN 3-540-19672-2.

[35] Les Hatton. Does OO Sync with How We Think? *IEEE Softw.*, 15(3):46–54, 1998. ISSN 0740-7459. doi: http://dx.doi.org/10.1109/52.676735.

[36] Mats P. Heimdahl. Verified Software: Theories, Tools, Experiments. chapter A Case for Specification Validation, pages 392–402. 2008. ISBN 978-3-540-69147-1.

[37] Constance Heitmeyer. On the Need for Practical Formal Methods. In *Formal Techniques in RealTime and Real-Time Fault-Tolerant Systems, Proc., 5th Intern. Symposium (FTRTFT'98)*, pages 18–26. Springer Verlag, 1998.

[38] Constance L. Heitmeyer, Ralph D. Jeffords, and Bruce G. Labaw. Automated Consistency Checking of Requirements Specifications. *ACM Trans. Softw. Eng. Methodol.*, 5: 231–261, July 1996. ISSN 1049-331X. doi: http://doi.acm.org/10.1145/234426.234431. URL `http://doi.acm.org/10.1145/234426.234431`.

[39] Peter B. Henderson. Mathematical reasoning in software engineering education. *Commun. ACM*, 46:45–50, September 2003. ISSN 0001-0782. doi: http://doi.acm.org/10.1145/903893.903919. URL `http://doi.acm.org/10.1145/903893.903919`.

[40] Erik Hofstee. *Constructing a Good Dissertation: A Practical Guide to Finishing a Master's, MBA or PhD on Schedule*. EPE, 2006. ISBN 0-9585007-1-1.

[41] IEEE Std 830-1998. IEEE Recommended Practice for Software Requirements Specifications. Technical report, IEEE, 1998. URL `http://ieeexplore.ieee.org/xpls/abs_all.jsp?arnumber=720574`.

[42] Jonathan Jacky. *The way of Z: practical programming with formal methods*. Cambridge University Press, New York, NY, USA, 1996. ISBN 0-521-55976-6.

[43] Wendy Johnston. A Type Checker for Object-Z. Technical report, Department of Computer Science, The University of Queensland Australia, 1996.

[44] Wendy Johnston and Gordon Rose. Guidelines for the Manual Conversion of Object-Z to C++. Technical report, Department of Computer Science, The University of Queensland Australia, 1993.

[45] Ivan Jureta, Stéphane Faulkner, and Pierre-Yves Schobbens. Clear justification of modeling decisions for goal-oriented requirements engineering. *Requir. Eng.*, 13(2): 87–115, 2008.

[46] Erik Kamsties, Daniel M. Berry, and Barbara Paech. Detecting Ambiguities in Requirements Documents Using Inspections. In *Proceedings of the First Workshop on Inspection in Software Engineering (WISE'01)*, pages 68–80, 2001.

[47] Jason Kealey and Daniel Amyot. Enhanced Use Case Map Traversal Semantics. In *SDL Forum*, pages 133–149, 2007.

[48] Jason Kealy. Enhanced Use Case Map Analysis and Transformation Tooling. Master's thesis, 2007. Ottawa-Carleton Institute for Computer Science.

[49] John C. Knight, Colleen L. DeJong, Matthew S. Gibble, and Lus G. Nakano. Why Are Formal Methods Not Used More Widely? In *4th NASA Formal Methods Workshop*, pages 1–12, 1997.

[50] Craig Larman and Victor R. Basili. Iterative and Incremental Development: A Brief History. *Computer*, 36:47–56, June 2003. ISSN 0018-9162. doi: 10.1109/MC.2003.1204375. URL `http://portal.acm.org/citation.cfm?id=972210.972226`.

[51] Yves Ledru. Complementing semi-formal specifications with Z. In *Proceedings of The 11th Knowledge-Based Software Engineering Conference*, pages 52–, Washington, DC, USA, 1996. IEEE Computer Society. ISBN 0-8186-7680-9. URL `http://portal.acm.org/citation.cfm?id=788008.788174`.

[52] David Lightfoot. *Formal Specification Using Z*. Grassroots Series. Palgrave, 2nd edition, 2001.

[53] Shaoying Liu. Internal consistency of FRSM specifications. *Journal of Systems and Software*, 29(2):167–175, 1995.

[54] Petra Malik and Mark Utting. CZT: A Framework for Z Tools. *ZB2005: Formal Specification and Development in Z and B, 4th International Conference of B and Z Users, Guildford, UK*, pages 65–84, 2005.

[55] Andrea Matta, Carlo A. Furia, and Matteo Rossi. Semi-formal and Formal Models Applied to Flexible Manufacturing Systems. In Cevdet Aykanat, Tugrul Dayar, and Ibrahim Korpeoglu, editors, *ISCIS*, volume 3280 of *Lecture Notes in Computer Science*, pages 718–728. Springer, 2004. ISBN 3-540-23526-4.

[56] Tim McComb and Graeme Smith. Animation of Object-Z Specifications Using a Z Animator. In *SEFM*, pages 191–, 2003.

[57] Steve McConnell. Feasibility Studies. *IEEE Software*, 15(3):119–120, 1998.

[58] W. McCune. Prover9 and Mace4. `http://www.cs.unm.edu/~mccune/prover9/`, 2005–2010.

[59] Andrew Miga. Application of Use Case Maps to System Design with tool Support. Master's thesis, Carleton University, 1998.

[60] Andrew Miga, Daniel Amyot, Francis Bordeleau, Donald Cameron, and Murray Woodside. Deriving Message Sequence Charts from Use Case Maps Scenario Specifications. In *Maps Scenario Specifications. 10th SDL Forum*, pages 268–287. Springer, 2001.

[61] Anne Miller. A work domain analysis framework for modelling intensive care unit patients. *Cogn. Technol. Work*, 6(4):207–222, 2004. ISSN 1435-5558. doi: http://dx.doi.org/10.1007/s10111-004-0151-5.

[62] David Mole. Z - An Introduction to Formal Methods, by Antoni Diller, Wiley, 2nd Edition, 1994 (Book Review). *Softw. Test., Verif. Reliab.*, 4(3):191, 1994.

[63] Ana Moreira and João Araújo. *Generating Object-Z Specifications from Use Cases*, pages 43–50. Kluwer Academic Publishers, Norwell, MA, USA, 2000. ISBN 0-7923-6239-X. URL `http://dl.acm.org/citation.cfm?id=343913.343932`.

[64] Gunter Mussbacher and Daniel Amyot. Goal and Scenario Modeling, Analysis, and Transformation with jUCMNav. In *ICSE Companion*, pages 431–432. IEEE, 2009. ISBN 978-1-4244-3494-7.

[65] Bashar Nuseibeh and Steve Easterbrook. Requirements engineering: a roadmap. In *Proceedings of the Conference on The Future of Software Engineering*, ICSE '00, pages 35–46, New York, USA, 2000. ACM. ISBN 1-58113-253-0. doi: 10.1145/336512.336523. URL `http://dx.doi.org/10.1145/336512.336523`.

[66] Gerard O'Regan. *Mathematical Approaches to Software Quality*. Springer-Verlag New York, Inc., Secaucus, NJ, USA, 2006. ISBN 184628242X.

[67] David Lorge Parnas. Tabular Representation of Relations. Technical report, 1992.

[68] Kasilingam Periyasamy and C. Mathew. Mapping a Functional Specification to an Object-Oriented Specification in Software Re-Engineering. In *CSC '96: Proceedings of the 1996 ACM 24th annual conference on Computer science*, pages 24–33, New York, NY, USA, 1996. ACM. ISBN 0-89791-828-2. doi: http://doi.acm.org/10.1145/228329.228331.

[69] Daniel Plagge and Michael Leuschel. Validating Z specifications using the PROB animator and model checker. In *IFM'07: Proceedings of the 6th international conference on Integrated formal methods*, pages 480–500, Berlin, Heidelberg, 2007. Springer-Verlag. ISBN 978-3-540-73209-9.

[70] Ben Potter, Jane Sinclair, and David Till. *An Introduction to Formal Specification and Z*. Prentice-Hall, Inc., Upper Saddle River, NJ, USA, 1991. ISBN 0-13-478702-1.

[71] Shengchao Qin and Guanhua He. Linking Object-Z with Spec#. In *ICECCS '07: Proceedings of the 12th IEEE International Conference on Engineering Complex Computer Systems*, pages 185–196, Washington, DC, USA, 2007. IEEE Computer Society. ISBN 0-7695-2895-3. doi: http://dx.doi.org/10.1109/ICECCS.2007.27.

[72] Jean-François Roy, Jason Kealey, and Daniel Amyot. Towards Integrated Tool Support for the User Requirements Notation. In *SAM*, pages 198–215, 2006.

[73] Robert G. Sargent. Verification and validation of simulation models. In *WSC '07: Proceedings of the 39th conference on Winter simulation*, pages 124–137, Piscataway, NJ, USA, 2007. IEEE Press. ISBN 1-4244-1306-0.

[74] Stephen R. Schach. *Object-Oriented and Classical Software Engineering*. McGraw-Hill, Inc., New York, NY, USA, 2008. ISBN 0073191264, 9780073191263.

[75] Ina Schaefer and Arnd Poetzsch-Heffter. Compositional Reasoning in Model-Based Verification of Adaptive Embedded Systems. In *SEFM '08: Proceedings of the 2008 Sixth IEEE International Conference on Software Engineering and Formal Methods*, pages 95–104, Washington, DC, USA, 2008. IEEE Computer Society. ISBN 978-0-7695-3437-4. doi: http://dx.doi.org/10.1109/SEFM.2008.16.

[76] Bran Selic. Using UML for Modeling Complex Real-Time Systems. In *Proceedings of the ACM SIGPLAN Workshop on Languages, Compilers, and Tools for Embedded Systems*, LCTES '98, pages 250–260, London, UK, 1998. Springer-Verlag. ISBN 3-540-65075-X. URL `http://dl.acm.org/citation.cfm?id=646905.710490`.

[77] Graeme Smith. *The Object-Z specification language*. Kluwer Academic, Boston, 2000. ISBN 0792386841.

[78] Graeme Smith. State-Based Formal Methods for Distributed Processing: From Z to Object-Z. Technical report, Software Verification Research Center, The University of Queensland Australia, 2001.

[79] Graeme Smith and John Derrick. Specification, Refinement and Verification of Concurrent Systems—An Integration of Object-Z and CSP. *Form. Methods Syst. Des.*, 18 (3):249–284, 2001. ISSN 0925-9856. doi: http://dx.doi.org/10.1023/A:1011269103179.

[80] Graeme Smith, Florian Kammller, and Thomas Santen. Encoding Object-Z in Isabelle/HOL. In *International Conference of Z and B Users (ZB 2002), volume 2272 of LNCS*, pages 82–99. Springer-Verlag, 2002.

[81] Colin Snook and Michael Butler. Using a graphical design tool for formal specification. In *Proceedings of the 13th Annual Workshop of the Psychology of Programming Interest Group*, pages 311–321, 2001.

[82] Ian Sommerville. *Software Engineering*. Addison-Wesley, 8th edition, 2007.

[83] Jim Mike Spivey. *The Z notation: a reference manual*. Prentice Hall International (UK) Ltd., Hertfordshire, UK, 1992. ISBN 0-13-978529-9.

[84] Susan Stepney, Fiona Polack, and Ian Toyn. Patterns to Guide Practical Refactoring: Examples Targetting Promotion in Z. In *International Conference of Z and B Users (ZB 2003)*, pages 20–39, 2003.

[85] Jing Sun, Jin Song Dong, Jing Liu, and Hai Wang. An XML/XSL Approach to Visualize and Animate TCOZ. In *Proceedings of the Eighth Asia-Pacific on Software Engineering Conference*, APSEC '01, pages 453–, Washington, DC, USA, 2001. IEEE Computer Society. URL `http://portal.acm.org/citation.cfm?id=872020.872430`.

[86] Chris Taylor, John Derrick, and Eerke Boiten. A Case Study in Partial Specification: Consistency and Refinement for Object-Z. In *Proc. of ICFEM 2000*, pages 177–185. IEEE, September 2000. URL `http://www.cs.kent.ac.uk/pubs/2000/1064`.

[87] Ian Toyn and John A. Mcdermid. CADiZ: An Architecture for Z Tools and its Implementation. *Software - Practice and Experience*, 25:305–330, 1995.

[88] Andrea Valerio, Giancarlo Succi, and Massimo Fenaroli. Domain analysis and framework-based software development. *SIGAPP Appl. Comput. Rev.*, 5(2):4–15, 1997. ISSN 1559-6915. doi: http://doi.acm.org/10.1145/297075.297081.

[89] Alta van der Merwe and Paula Kotzé. Criteria used in selecting effective requirements elicitation procedures. In *SAICSIT '07: Proceedings of the 2007 annual research conference of the South African Institute of Computer Scientists and Information Technologists on IT research in developing countries*, pages 162–171, New York, NY, USA, 2007. ACM. ISBN 978-1-59593-775-9. doi: http://doi.acm.org/10.1145/1292491.1292510.

[90] John Andrew van der Poll. *Automated support for set-theoretic specifications*. PhD thesis, School of Computing, University of South Africa (South Africa), 2000. Promoter-Kotze, P. and Promoter-Labuschagne, W. A.

[91] John Andrew van der Poll and Paula Kotzé. What design heuristics may enhance the utility of a formal specification? In *SAICSIT '02: Proceedings of the 2002 annual research conference of the South African institute of computer scientists and information technologists on Enablement through technology*, pages 179–194, , Republic of South Africa, 2002. South African Institute for Computer Scientists and Information Technologists. ISBN 1-58113-596-3.

[92] John Andrew van der Poll and Paula Kotzé. A multi-level marketing case study : specifying forests and trees in Z. *South African Computer Journal*, 30:17–28, 2003.

[93] John Andrew van der Poll and Paula Kotzé. Enhancing the Established Strategy for Constructing a Z Specification. *SACJ*, (No. 35):118–131, 2005.

[94] John Andrew van der Poll, Kotzé Paula, Ahmed Seffah, Thiruvengadam Radhakrishnan, and Asmaa Alsumait. Combining UCMs and Formal Methods for Representing

and Checking the Validity of Scenarios as User Requirements. *SAICSIT'03*, pages 111–113, 2003.

[95] Axel van Lamsweerde. Formal specification: a roadmap. In *Proceedings of the Conference on The Future of Software Engineering*, ICSE '00, pages 147–159, New York, NY, USA, 2000. ACM. ISBN 1-58113-253-0. doi: http://doi.acm.org/10.1145/336512.336546. URL `http://doi.acm.org/10.1145/336512.336546`.

[96] Axel Van Lamsweerde. Goal-Oriented Requirements Engineering: A Guided Tour. In *RE '01: Proceedings of the Fifth IEEE International Symposium on Requirements Engineering*, page 249, Washington, DC, USA, 2001. IEEE Computer Society.

[97] Roel Wieringa, Eric Dubois, and Sander Huyts. Integrating semi-formal and formal requirements. In *Proceedings of the 9th International Conference on Advanced Information Systems Engineering*, pages 19–32, London, UK, 1997. Springer-Verlag. ISBN 3-540-63107-0. URL `http://dl.acm.org/citation.cfm?id=646085.679588`.

[98] Kirsten Winter and Roger Duke. Model Checking Object-Z Using ASM. In *Proceedings of the Third International Conference on Integrated Formal Methods*, IFM '02, pages 165–184, London, UK, UK, 2002. Springer-Verlag. ISBN 3-540-43703-7. URL `http://dl.acm.org/citation.cfm?id=647983.743685`.

[99] Jane Wood and Denise Silver. *Joint application development (2nd ed.)*. John Wiley & Sons, Inc., New York, NY, USA, 1995. ISBN 0-471-04299-4.

[100] Jim Woodcock. Calculating Properties of Z specifications. *SIGSOFT Softw. Eng. Notes*, 14(5):43–54, 1989. ISSN 0163-5948. doi: http://doi.acm.org/10.1145/71633.71634.

[101] Jim Woodcock and Jim Davis. *Using Z: Specification, Refinement, and Proof*. Prentice-Hall, Inc., Upper Saddle River, NJ, USA, 1996. ISBN 0-13-948472-8.

[102] Yong Xiang Zeng. Transforming Use Case Maps to the Core Scenario Model Representation; http://lotos.site.uottawa.ca/ftp/pub/Lotos/Theses, Juin 2005.

[103] Jianjun Zhao and Baowen Xu. Measuring Aspect Cohesion. In Michel Wermelinger and Tiziana Margaria, editors, *FASE*, volume 2984 of *Lecture Notes in Computer Science*, pages 54–68. Springer, 2004. ISBN 3-540-21305-8.

Index