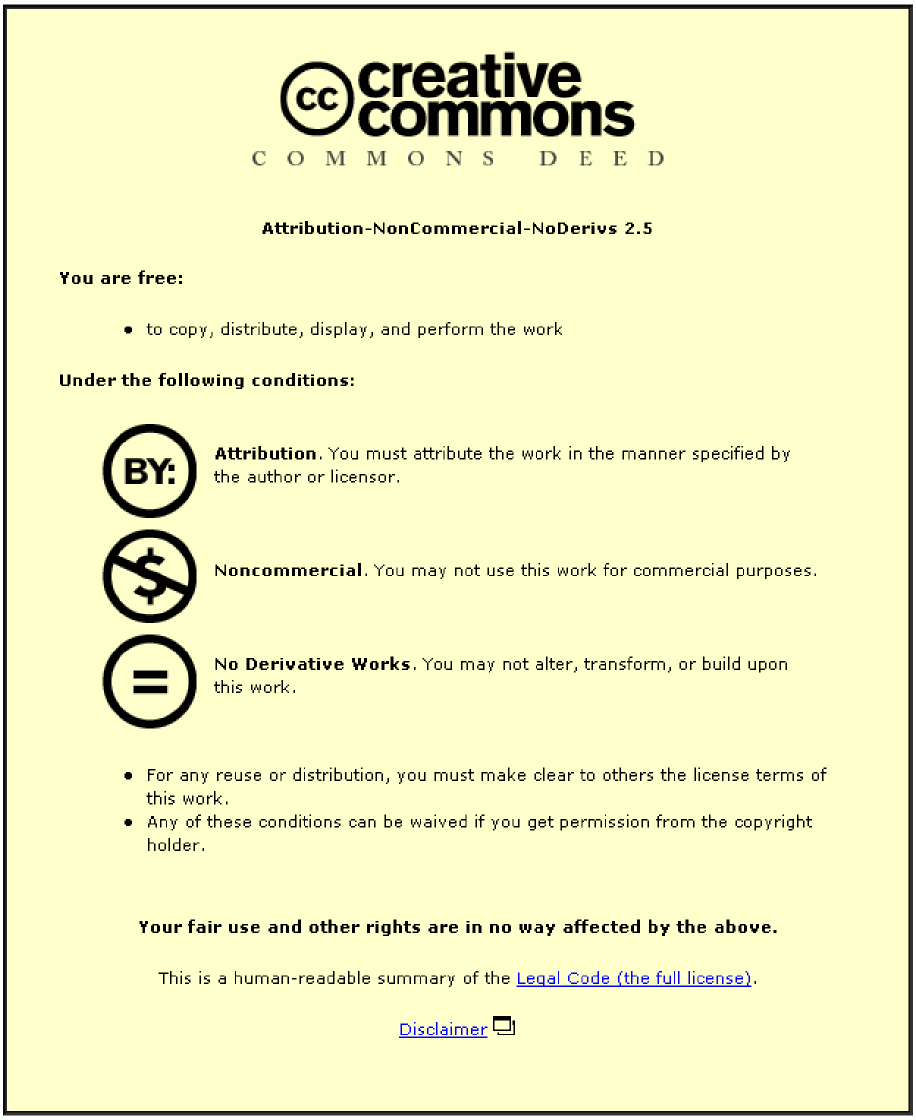
Requirements specification using concrete scenarios



Requirements Specification Using Concrete Scenarios

by

Oliver T.S. Au

A Doctoral Thesis

Submitted in partial fulfilment of the requirements
for the award of

Doctor of Philosophy

of

Loughborough University

Friday 18th September 2009

ABSTRACT

The precision of formal specifications allows us to prove program correctness. Even if formal methods are not used throughout the software project, formalisation improves our understanding of the problem. Formal specifications are amenable to automated analysis and consistency checking. However using them is challenging. Customers do not understand formal notations. Specifiers have difficulty tackling large problems. Once systems are built, formal specifications quickly become outdated during software maintenance. A method of developing formal specifications using concrete scenarios is proposed to tackle the disadvantages just mentioned.

A concrete scenario describes system behaviour with successive steps. The pre- and post-states of scenario steps are expressed with actual data rather than variables. Concrete scenarios are expressed in a natural language or formal notation. They increase customer involvement in the creation of formal specifications. Scenarios may be ranked by priorities allowing specifiers to focus on a small part of the system. Formal specifications are constructed incrementally. New requirements are also captured in concrete scenarios which guide the modification of formal specifications.

On one hand, concrete scenarios assist the creation and maintenance of formal specifications. On the other hand, they facilitate program correctness proofs without using conventional formal specifications. This is achieved by adding implementation details to customer scenarios. The resulting developer scenarios, encapsulating decisions of data structures and algorithms, are generalised to operation schemas. With the implementation details, the schemas written in formal notations are programs rather than specifications.

Keywords: Concrete Scenarios, Formal Methods, Software Specifications, Requirements Elicitation, Software Maintenance, Nondeterminism, Scenario Expansion, Specification Refinement

ACKNOWLEDGEMENTS

I am grateful for my family. My parents Ying Fat and Wai Fong paid for the first four years of my education in Canada. They worked very hard to make ends meet. Their love for me is boundless. My wife Cudie encouraged me when I was an unmotivated student until I eventually turned around. My son Jasper kept me cheerful throughout my PhD research which could otherwise be gruesome at times.

Minoru Hasegawa is an emeritus professor at Lakehead University in Thunder Bay, Ontario, Canada. He developed in me a keen interest in computer programming. He showed that passion is an alternative to competition in the pursuit of excellence.

George Tourlakis of York University in Toronto taught me more about computing than any other individual. He encouraged me to go to graduate school.

Ric Hehner was my M.Sc. supervisor at the University of Toronto. He proved that an outstanding scholar need not be arrogant or pretentious. Ric set me a very good example of a researcher and a teacher.

John Cooke was a key reason that I had chosen to go to Loughborough for my PhD research. After his early retirement, John continued to supervise my research. His insights had a great influence in the direction of my research.

Roger Stone has become my main supervisor after John's retirement. Roger's attention to details is a great asset. He has been helpful in every aspect of my research. During my visits to Loughborough, Roger and his wife Jacqueline allowed me to stay in their house whenever the university was unable to offer me a temporary accommodation. Without Roger's help, my learning experience in UK would have been less fruitful and less enjoyable.

Marian Petre provided helpful comments to the first submission of this thesis. Meng Heng, Lam and Kain were my subjects in the empirical study.

CONTENTS

LIST OF FIGURES

LIST OF TABLES

1. INTRODUCTION

Requirements specifications and their suport of program derivation have been a challenge to software developers since the beginning of the computer era. We introduce and advocate the use of concrete scenarios to create formal requirements specifications and to support program development. Section 1.1 discusses the desirable qualities of requirements specifications and their relationships with programs. Section 1.2 explains the challenges posed by formal specifications. Section 1.3 gives an example of concrete scenarios but we will wait until Chapter 3 to use them to derive a formal specification in the Z notation. The derivation of Java programs from concrete scenarios will appear in Appendix I. Section 1.4 lists our contributions. Sections 1.5 and 1.6 justify our research methodology and choice of examples. Section 1.7 reviews our program view. Section 1.8 outlines the dissertation.

1.1 Requirements Specifications

Our work concerns the effective transfer of information from customers to developers so that correct programs can be written. We need to clarify a few terms for a meaningful discourse.

Requirements are about the application domain not the machine [84]. A **program** is about the machine phenomena which are the responsibility of programmers [84]. A **specification** documents an agreement between the customer and the developer [58] [84]. There are at least two kinds of specifications. A *requirements specification* embodies customer requirements; a *design specification* contains design decisions made by the developers [42]. We are interested in the requirements specification which bridges between the requirements and the program.

We define a **customer** as someone who pays for the program or uses it.

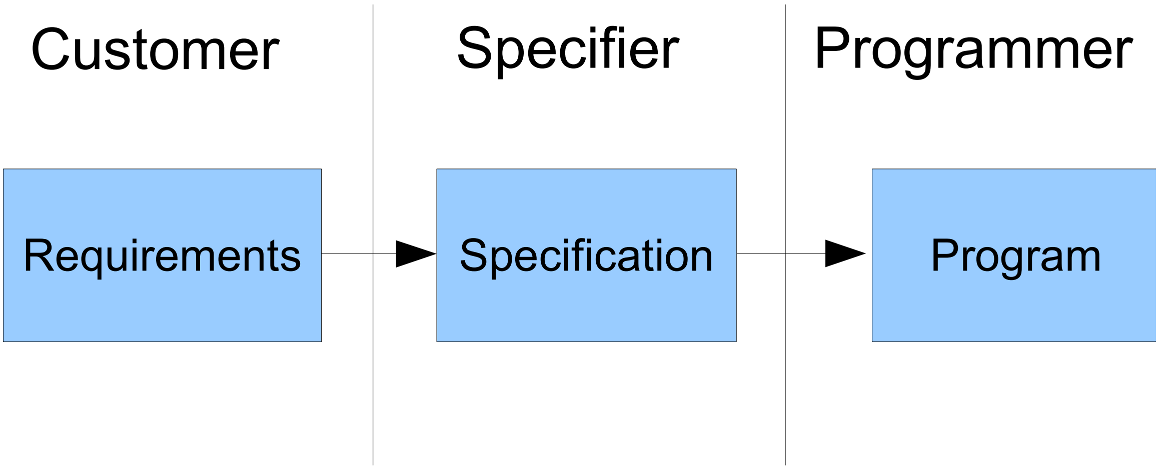


Fig. 1.1: A simplistic view of software development

He or she is the source of software requirements. A **specifier** is responsible for the creation of the specification capturing the requirements. A **programmer** creates a program with reference to the specification. In Chapter 7, we will see that the distinction between a specification and a program is not always clear-cut. When it is not necessary for us to identify an artefact as a specification or a program, we refer to its creator as a **developer** rather than a specifier or a programmer.

1.1.1 Desirable Quality

It is challenging to write a good specification. The requirements tend to be vague when first elicited from the customer. The program, on the other hand, must be exact to be recognisable by the compiler for the generation of executable code. To facilitate the creation of a satisfactory program, the IEEE Standard 830-1998 *Recommended Practice for Software Requirements Specifications* states that a good software requirements specification (SRS) should have the following characteristics.

1. **Correct** - Requirements are accurately captured to reflect customers'

needs.

2. **Unambiguous** - Every requirement has only one interpretation.
3. **Complete** - All requirements, relating to functionality, performance, design constraints, attributes, or external interfaces, are included. Valid and invalid input values are specified with expected responses.
4. **Consistent** - No subsets of individual requirements conflict internally or with real-world objects. There is only one name for one object.
5. **Ranked for importance and/or stability** - When requirements are ranked for importance, the ranks include though are not limited to *essential*, *desirable* and *optional*. The stability of a requirement may be captured by the number of expected changes.
6. **Verifiable** - For every requirement, there exists a cost-effective manual or automated process to check its fulfilment by the program.
7. **Modifiable** - Each requirement is written separately without mixing with other requirements. Redundancy is minimised and allowed only for improved readability. There are table of contents, an index and cross-references to facilitate modifications.
8. **Traceable** - Each requirement explicitly references its source in ealier documents for *backward traceability*. Each requirement also has a unique name or reference number for *forward traceability*.

Leffingwell and Widnig add the desirable quality of **understandability** [100]. A requirement is understandable if it is fully comprehended by the customer and the developer. These nine characteristics may be used in an evaluation of specification languages or approaches.

1.1.2 Relationships with Programs

A requirements specification describes features essential to the customer. Details in the program, of no concern to the customer, are supplied by the programmer. Different programs may meet the same set of requirements.

During *verification*, the programmer or tester determines whether a program meets the requirements specification. In this exercise, we ask, “Are we building the product right?” [20, page 37]

Specifications may contain errors. Programs based on erroneous specifications are also incorrect. The exercise of determining a specification’s correctness is called *validation*. Validation is a set of activities to ensure that the program is traceable to customer requirements [119, page 467]. In other words, we ask, “Are we building the right product?”

1.2 Challenges in Using Formal Specifications

Our research is motivated by three phenomena regarding the usage of formal specifications. The first is the lack of customer involvement. According to the Standish group’s CHAOS report, the most important success factor is customer involvement [61]. Formal specifications are written by formalism experts based on their understanding of the application domains which may differ from the customers’ expectations. Customers cannot provide feedback on the correctness of the formal specifications due to the arduous notations used [149]. Time and effort are wasted until the mistaken requirements are finally caught by testing. The limited adoption of formal methods should not be surprising [59]. Through the use of concrete scenarios, we hope to improve customer involvement in the creation of formal specifications.

Secondly, the formal specification literature focuses on suitable sets of notations and analysis of specifications. There is a general lack of guidance for producing an initial formal specification from requirements [115]. Instead of inventing new formal specification languages, van Lamsweerde advises more effort be put into devising methods for the creation and modification of good specifications [140]. We devise an approach to guide the specifiers so that formal specifications can be built one step at a time.

Lastly, we aim to extend the use of formal specifications beyond initial development. Specifications must keep up with changing requirements. The need for specification maintenance was noted by Bustard and Winstanley [33]. Previous research has been limited to formal specifications of event ordering [95][139]. Our objective is to devise an iterative formal methodology

Step	**Interaction**	**System State**
0		No. 4 is connected to no. 5.
1	Caller at no. 6 dials to no. 7. Caller gets a ring tone.	No. 4 is connected to no. 5. No. 6 is ringing no. 7.
2	Callee at no. 7 answers the phone. Request is OK.	No. 4 is connected to no. 5. No. 6 is connected to no. 7.
3	User at no. 6 hangs up the phone. Request is OK.	No. 4 is connected to no. 5.

Tab. 1.1: **E-scenario** *MakeSimpleCall*

that can cope with evolving requirements for general computation.

1.3 A Simple Concrete Scenario

With a number of steps, a **concrete scenario** documents how a user task is performed in a particular situation. A **scenario step** is described by a pre-state, a post-state, and some input/output parameters. Table 1.1 is a concrete scenario of making a telephone call written in English. The stilted writing style facilitates its translation to a formal notation. Single-digit telephone numbers are used for their compactness. There are three steps in this scenario. The pre-state of step 1 is shown on row 0 where phone number 4 is already connected to phone number 5. This connection does not change throughout the scenario. In step 1, a user at phone number 6 dials phone number 7 and he gets a ring. The post-state of this step is shown in the right on the same row where phone number 6 is ringing phone number 7. The post-state of step 1 is also the pre-state of step 2. In step 2, the user at phone number 7 answers the phone. A connection of numbers 6 and 7 is made. In step 3, the user at phone number 6 hangs up. The connection is terminated.

Table 1.2 on the following page is an equivalent concrete scenario expressed in Z notation. In later chapters, we will explain the process of writing concrete scenarios in English and in Z.

Concrete scenarios written in English are *E-scenarios* allowing customers to confirm the required functionalities in a language that they understand.

Step	**Interaction**	**System State**
0		$connection = \{4 \mapsto 5\}$
1	$caller? = 6 \land callee? = 7$ $tone! = ring$	$connection = \{4 \mapsto 5\}$ $ringing = \{6 \mapsto 7\}$
2	$answer? = 7$ $rqt! = OK$	$connection = \{6 \mapsto 7, 4 \mapsto 5\}$
3	$hang? = 6$ $rqt! = OK$	$connection = \{4 \mapsto 5\}$

Tab. 1.2: **Z-scenario** *MakeSimpleCall*

Concrete scenarios written in Z are *Z-scenarios* which allow specifiers to generalise steps to Z schemas accurately. Concrete scenarios are not written in a new language but in a small subset of a chosen formalism. For example, the Z-scenario above uses a subset of Z notation. We have chosen Z to illustrate concrete scenarios for two reasons. First, it is widely known to the formal method community. Second, it has schema operators for incremental construction of formal specifications.

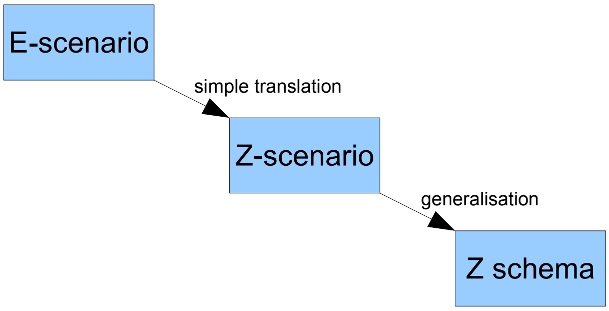


Fig. 1.2: Creating Z schemas from concrete scenarios

Concrete scenarios are first written in English, then translated to their equivalent concrete scenarios in a chosen formalism, and finally generalised to a formal specification as shown in Figure 1.2.

Scenario is a popular term. It has been used for many things. We call ours *concrete scenario* to distinguish it from other usage. When it is clear in the context that we are referring to *concrete scenarios*, we may drop the word *concrete*.

1.4 Contributions

The work is based on two premises. First, formal specifications are not understandable to customers [51][112]. Second, informal specifications are too ambiguous to be useful for reliable creation of formal specifications [84][100].

Our first contribution is a way of representing scenarios by a sequence of steps expressed with actual data rather than variables. The scenarios are precise enough to guide the development of formal specifications and at the same time more accessible to customers. Through concrete scenarios, customers can participate in the creation of formal specifications.

Our second contribution is an approach to create formal specifications iteratively from concrete scenarios. Specifiers incrementally construct formal specifications following the priorities of the customer. Formal specifications can be kept up-to-date with reasonable effort.

Our third contribution is an approach to create programs from concrete scenarios. Scenarios are originally written from customers' perspective and then expanded to include implementation details that concern developers. The expanded scenarios facilitate communications that concern developers.

Concrete scenarios are good for the verification of specifications and programs making them the ultimate reference of customer requirements.

1.5 Research Philosophy

We have to be careful with the wording of our claims lest we make the mistake that Rugg and Petre have warned us about [123, Page 40]. We cannot prove that all integers are even numbers by enumerating a long list of them, such as 2, 4, 6, 8 and so forth. By the same token, we cannot claim that the use of concrete scenarios will always be beneficial by showing a number of examples in which they shine.

We claim that concrete scenarios can be used to elicit user requirements so that requirements can be faithfully captured in a formal specification. We further claim that concrete scenarios support iterative revisions of a formal specification. Concrete scenarios can capture scenarios precisely from the developer's perspective as well as the customer's perspective.

We support these claims by applying concrete scenarios to application domains with varying characteristics. We only manage to show that our scenario-driven approach is doable without any clear hint of exception. We cannot prove or be expected to prove that the use of concrete scenarios are always beneficial in all application domains.

Despite the stated limitation in our claims, we formally define concrete scenarios so that they can be used to validate a formal specification. This is a significant part of the dissertation. As long as software customers find concrete scenarios more understandable than formal specifications, concrete scenarios are poised to improve the appeal of formal methods to a larger audience.

The usefulness of concrete scenarios is dependent on the background of the developer. A formalism expert may well have a bias for skipping concrete scenarios to directly create a formal specification. It is out of our current scope to find out how people from different backgrounds respond to the use of concrete scenarios.

1.6 Choice of Examples

In our exploration, we have considered four application domains. The choice are made so that a wide range of characteristics are covered. It increases our comfort level that the approach is applicable and beneficial in other domains.

The first example is a warehouse ordering problem. The author tackled it at the time he had little Z experience. We used a published Z specification as our example so that our solution can be compared for correctness. In retrospect, we realise another advantage. There can be a large number of products in an order or a warehouse. It is only necessary to specify a small number of products in our scenarios. Once generalised to operation schemas, arbitrarily large numbers of products can be handled. The example demonstrates the power of generalisation.

The second example is a telephone switching system. It was chosen for its high interactivity. A scenario has multiple steps involving several users. We start with the basic functionality and add conferencing to the completed specification to test the efficacy of our approach in specification maintenance.

Unlike other examples, the sorting problem is purely computational. With no user interactions, it is a one-step operation to the customer. Yet, it consists of many steps to the developer. We explore how the approach deals with the differing perspectives of the customer and the developer. The example sets the stage to discuss the relationship between a program and a specification.

The last example is dice rolling. We express nondeterministic behaviour using scenarios and generalise them to a nondeterministic specification. With the help of scenario expansion introduced in the sorting example, we write a deterministic program to simulate the random behaviour. In the examples of sorting and dice rolling, we create verifiably correct programs from scenarios without going through formal specifications.

1.7 Program View

In the creation of Z specifications from scenarios, we adopt the view that a program is a collection of operations $OP_1, OP_2, \ldots, OP_m$. Each operation is defined as a disjunction of a number of schemas $h_1, h_2, \ldots, h_n$ where a schema h_j handles steps of scenarios meeting a certain precondition.

$$OP_i \mathrel{\widehat=} h_1 \lor h_2 \lor \cdots \lor h_n$$

The schemas $h_1, \ldots, h_n$ are grouped into an operation by the signature they share. The signature of a schema is defined by the names and types of its input and output parameters.

The Z notation allows the details of a specification to be written at two levels. At the low level inside a schema, there are operators to build predicate expressions from simpler predicates. At the high level, there are operators to construct schema expressions from simpler schemas. The same notions are available once at the predicate level and again at the schema level. The redundancy is more for convenience than necessity. We restrain ourselves to using the disjunct schema operator when defining operations. It simplifies our program view without sacrificing expressive power.

1.8 Outline

Chapter 2 discusses other work on requirements specification. In Chapter 3, we introduce concrete scenarios and create a formal specification of a warehouse ordering system showing that concrete scenarios afford customer involvement in formal specification. In Chapter 4, we specify a telephone system. The focus is on providing guidance to specifiers. In Chapter 5, we maintain a formal specification by considering new scenarios which demand an update to the data structure. Chapter 6 formally defines the observance relation between Z schemas and concrete scenarios. In Chapter 7, customer scenarios for the sorting problem are expanded to developer scenarios to include implementation details. Chapter 8 uses a dice rolling simulator to explore nondeterminism. Chapter 9 concludes our work.

2. A SURVEY OF REQUIREMENTS SPECIFICATION TECHNIQUES

The chapter surveys both formal and informal requirements specification approaches. A formal method consists of formal specification and design calculi techniques [18]. A *formal specification* is expressed in a mathematical language that has precise syntax and semantics. A specification written in a mathematical language can be analysed for consistency. A *design calculus* is a set of proof rules or transformation rules. Proof rules can be used to prove a program correct with respect to a formal specification. Transformation rules can be used to refine a formal specification step-by-step to a program that is guaranteed to be correct.

Our discussion starts at the informal end of the spectrum. Use cases are a popular informal specification approach in which user tasks are described as sequences of steps in the user's language. Test-driven development (TDD) guides programming with test cases. The executable tests, normally unit tests, are written before the actual programs to give programmers a better understanding of the requirements. Though not a specification approach per se, TDD inspired our use of concrete scenarios in the writing of specifications.

Structured analysis and object-oriented analysis are intuitive due to their graphical notations. Their relatively shallow learning curve allows more people to take part in the requirements activities. They are only considered semi-formal because of the lack of mechanical ways to prove one way or the other that two graphs are equivalent. the only their syntax is formally defined but not their semantics.

Petri nets have an uncommon advantage of being both graphical and formal. However the basic Petri nets would be quite awkward to use in dealing with general problems. The bells and whistles in advanced variants of Petri nets make the approach more usable but at the same time demand more training from the writers and readers.

We go on to discuss a few representative formal notations: the model-based Vienna Development Method Specification Language (VDM-SL), the event-based language of Temporal Ordering Specification (LOTOS), an algebraic language Larch and a tabular language called Software Cost Reduction (SCR).

In a rule-based approach, rules can be written as low-level production rules or high-level business rules. The abstract state machine (ASM) has the flexibility to begin at the high-level and be gradually refined to the low-level for implementation.

We also discuss hybrid approaches. One approach combines Data Flow Diagrams (DFDs) used in Structured Analysis with VDM. Structured-object-based-formal language (SOFL) is a variant of the above combination with an added flavour of object-orientation. Framework for integrated test (Fit) is a test-driven tabular programming approach. Controlled natural language uses a subset of natural language with a formal underpinning.

2.1 Use Cases

An *actor* is a person or another system that interacts with the system we are building. A *primary actor* requests our system to perform a task in order to achieve a goal. The task may require the collaboration of *secondary actors*. A use case is a description of the actions to accomplish the task. Multiple actors may participate in a use case. Following is a sample use case we have adapted [126, page 36]. It may be described in another format with additional fields. The four most important fields are shown here: use case name, precondition, flow of events and postcondition.

Table 2.1 is a use case described in low-level details. A use case may also be described at a higher-level. It may refer to low-level use cases [38, page 206]. For example, we may extend a main use case with an exception-handling use case so that the event flow in the main use case is not cluttered by exceptional events.

The events in the use case description were written in simple sentences with clear subjects, verbs and objects to reduce ambiguity. However natural

Use Case: Place Order
Precondition: A customer has logged on.
Flow of Events:
1. The customer selects Place Order.
2. The customer enters his or her name and address.
3. The customer enters product codes and quantities for items of his or her choice.
4. The system supplies a product description and price for each item.
5. The system keeps a running total of items ordered.
6. The customer enters credit card information.
7. The customer selects Submit.
8. The system verifies the information and saves a pending order.

Postcondition: The new pending order is saved on the system.

Tab. 2.1: A simple use case

languages are inherently ambiguous. The events normally assume a simple order. For example, event 4 follows event 3. But it is unclear if repetitions are allowed. Some practitioners use conditionals and loops to describe use cases more precisely [126, page 25] but this is a controversial practice. It makes the use case harder to read. The added control information may be preferred by the programmers but probably not by the customers.

Use cases are understandable without special training to anyone who speaks the language. The software community at large embraces use cases as an effective means to describe functional requirements. In summary, use cases fare well in four of the nine desirable specification characteristics. They are understandable, modifiable and traceable and can readily be ranked for importance and stability. They are mediocre in the other five characteristics.

A scenario is a task performed in a particular situation. A use case consists of several scenarios; some successfully achieve the goal and others do not. A scenario description often looks like a use case description. The boundary between them is not clear-cut. What one person chooses to call a scenario, another may call a use case.

There is a difference between scenarios and our concrete scenarios. A scenario describes a family of instances. Following is a scenario written in a simple though uncommon format.

A bank customer comes to an ATM. He uses his ATM card

> number and password to withdraw money. The transaction ends after the ATM machine dispenses the money, a receipt and the ATM card.

A concrete scenario describes a particular instance. It contains actual data of the card number, password and amount withdrawn. Here is an example of a concrete scenario in English written as prose.

> George Harrison is a customer of the HSBC. He inserts his ATM card number 12345678 to access the ATM number 9421. He enters password "229345" when prompted. He selects withdrawal and keys in 80. The ATM machine dispenses $80, a receipt and the ATM card. George takes them and leaves.

2.2 Test-Driven Development (TDD)

Test-driven development (TDD) is a practice of Extreme Programming (XP) [14]. TDD guides programmers with small test cases. It is an iterative development approach with each iteration consisting of five tiny steps [15].

1. write a test
2. make it compile
3. run it to see if it fails
4. modify the code until the test succeeds
5. refactor the code to remove any duplication introduced

Both TDD and XP belong to a family of agile methodologies. Their premise is that requirements are subject to change. To reduce effort wasted on creating and updating documents with little value, they try to get on with the programming task with minimal documentation and planning. The agile community by and large does not care for the use of formal methods. Baumeister is a notable exception. He extends TDD by adding two new steps to the end of a TDD iteration [13].

6. generalize the tests to add assertions to code

7. refactor assertions

The approach was illustrated with Java Modeling Language (JML). It addresses a shortcoming of TDD that test cases, being just examples, do not completely describe the program behaviour. The objective is to gain some benefits of formal specifications in the context of TDD while keeping the agility. Written after the program code, the assertions do not guide the programmer in the coding activities. Yet they still help the programmer to acquire a better understanding of the program.

2.3 Structured Analysis (SA)

In the early era of computing, analysis and programming were conducted with ad hoc methods. A search for disciplined, well-thought-out methods resulted in a collection of structured techniques [105, page 3]. In that light, SA is not confined to any specific method or notation. However the most popular artefacts in SA are Entity-Relationship Diagrams (ERD), Data Flow Diagrams (DFD) and State-Transition Diagrams (STD) [119, Chapter 12].

Data modelling identifies data objects, their attributes, relationships, cardinality and modality. Cardinality is the maximum number of objects allowed in a relationship. Modality indicates if a related object is mandatory.

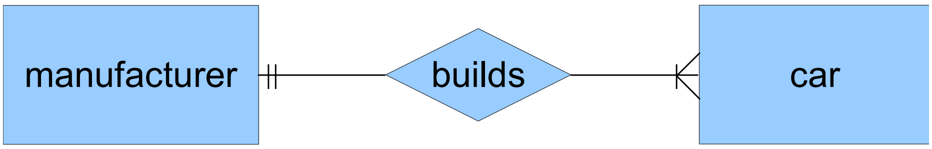


Fig. 2.1: A simple ERD

Figure 2.1 is a simple ERD that describes a one-to-many mandatory relationship between a manufacturer and a car [119, Page 301]. A function is represented by a bubble in a DFD. The functional modelling begins with a level 0 DFD which has only a single bubble (function). There are graphical symbols to represent external entities and data stores. Every bubble in a DFD is refined into a complete DFD at the next level until the desired level of detail is reached. Information flow is expressed with an arrow. All bubbles and arrows should be labelled. Behavioural modelling identifies the possible states of a system in an STD. An arrow connecting two states represents a state transition. In structured analysis, ERDs, DFDs and STDs are created in distinct activities [49]. The separation of data modelling and behavioural modelling does not lead to a system structure that is easy to maintain. Object-oriented analysis was proposed to address those shortcomings.

2.4 Object-Oriented Analysis and Unified Modelling Language

Rumbaugh et. al. define an object as a concept, abstraction, or thing with crisp boundaries. They use objects to represent the real world and to provide a practical basis for programs [124]. The Unified Modeling Language (UML) is by far the most common OO notation. It originated as an amalgamation of the different notations advocated by Booch, Jacobson and Rumbaugh [23]. There were many diagram types in the original UML to represent different aspects of a system. Widely used UML diagrams are:

1. Use case diagram
2. Class diagram
3. Object diagram
4. Sequence diagram
5. Collaboration diagram
6. Statechart diagram
7. Activity diagram

Some information in one diagram may be duplicated in a diagram of a different type. The sequence diagram holds the same information as the col-

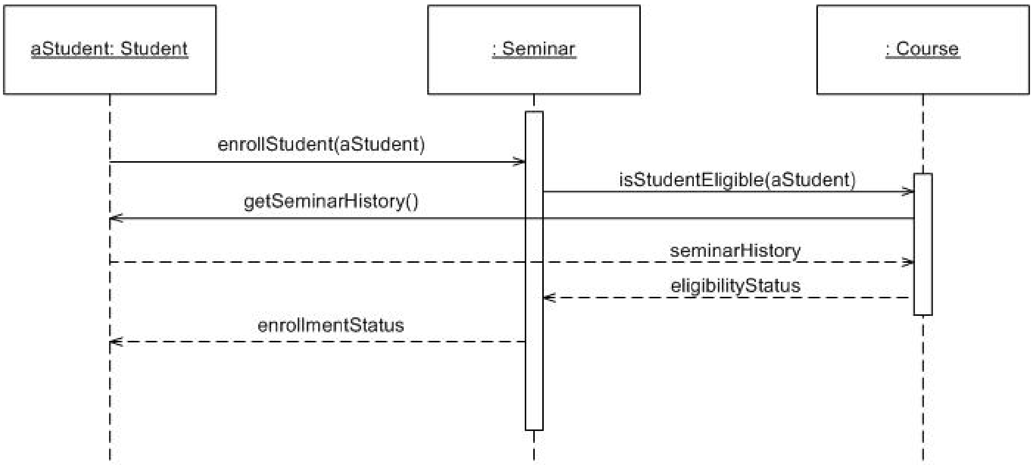


Fig. 2.2: A sequence diagram

laboration diagram only presented differently. Good tool support is required to check inconsistency across diagrams.

The sequence diagram in Figure 2.2 on the next page has three objects of classes *Student*, *Seminar* and *Course* respectively [5]. Horizontal arrows denote messages which are labelled with message names and parameters.

The UML is too rich for practitioners to master all its intricacies. Only a subset of the UML is used for most applications. Its graphical notation is both a blessing and a curse. Customers are well aware of the fact that they do not understand formal specifications. But they would always try to make sense of graphical models. Their guess could be wrong. Intuitive graphical notations often give people the false impression of effective communication. Ambiguities hide behind the attractive diagrams.

Large corporations have invested heavily in UML. It is unclear whether their investments are driven by the merits of UML or the fear of being left behind. It is doubtful if the corporations, other than UML tool vendors, have received a reasonable return on their investments.

2.5 Petri Nets

Petri nets are a major model of concurrent systems [11]. Despite the graphical representation, they are formal. A Petri net consists of *places* shown

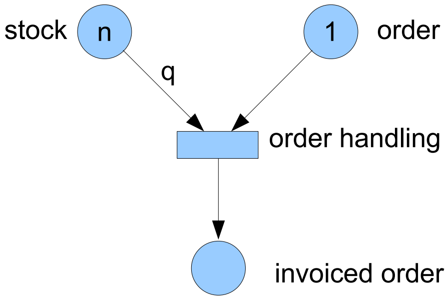


Fig. 2.3: A Petri net

as circles and *transitions* shown as rectangles. Figure 2.3 is a Petri net of a warehouse order handling operation. Transition *order handling* has two input places *stock* and *order* and one output place *invoiced order*. Place *stock* has n tokens representing the quantity of an item in stock. Place *order* has one token representing a pending order. Label q on an arc represents the ordered quantity. If $n \geq q$, the transition of order handling is fired resulting in a token added to place *invoiced order* representing an invoiced order. If the number of tokens is small and known, we usually use one black dot to represent each token.

A transition is enabled to fire if tokens are available in all its input places. The firing of a transition removes one token from each input place and adds one token to each output place. The default of one token can be overriden by labelling an arc with a positive integer as in the previous Petri net.

The state of a Petri net is represented by a *marking* that shows the number of tokens in each place. For example, the previous Petri net has a marking of $(n \quad 1 \quad 0)$. After the transition is fired, the numbers of tokens in the two input places will be decremented by q and 1 respectively. The number of tokens in the output places will be incremented by 1. The updated marking will be $(n - q \quad 0 \quad 1)$. When multiple transitions are enabled at the same time, any of the transitions may fire. The choice of the transition to fire is made in a nondeterministic manner, i.e., randomly or by forces that are not modelled [114].

Bernardinello and de Cindio classify Petri nets into three levels [17]. Petri nets in the first level are characterised by Boolean tokens. Places can only be marked by at most one token. Condition/Event (C/E) Petri nets belong to level 1. Petri nets in the second level are characterised by multiple tokens in one place serving as counters. Place/Transition (P/T) Petri nets belong to level 2 of which our warehouse ordering Petri net is an example. Petri nets in the third level are characterised by structured tokens holding additional information beyond the counters. Coloured Petri nets belong to level 3 [88].

Our warehouse ordering Petri net only handles orders of a particular item for a fixed quantity. It does not handle the ordering of multiple items for arbitrary quantities. Petri net extensions at level 3 can help express requirements more elegantly but they may make the representation harder to understand by the customers.

Petri nets can be analysed to answer useful questions about a model, for example, whether a Petri net can deadlock. If tokens represent resources, we can analyse the net to see if the resources are conserved after transitions. Many questions can be reduced to reachability problems which ask if initial markings lead to specific resulting markings.

2.6 VDM: a Model-based Language

A model-based specification is also called state-based. It is expressed as a state model using common mathematical entities such as sets and functions [131, Chapter 10]. An operation is defined by its effects on the system state in terms of the relationship of variable values before and after the operation.

Z [144], B [2] and VDM-SL [92] are model-based specification languages. The Z notation is based on set theory and predicate logic. The closely related B-Method supports the complete software life cylce from specification to implementation using B-Tool and Abstract Machine Notation (AMN). VDM stands for Vienna Development Method; SL stands for Specification Language. People often use VDM to mean VDM-SL. Figure 2.4 on page 35 is a VDM specification for the hotel room booking operation [3, page 220].

In block **types**, we define type *RoomNumber* for values from 1 to 100 and type *RoomStatus* for available or occupied. In block **state**, we define variable *rooms* as a map from room number to room status. All rooms are available initially. In block **operations**, the *book-room* operation is defined with a room number parameter. The second statement in the block declares that the external variable *rooms* is accessed for read and write. The precondition, denoted by keyword **pre**, is that the parameter is a valid room number and the room is available. In the original publication, the second conjunct of the precondition about room availability is missing. The version you see here has been corrected. Using override symbol †, the postcondition specifies that the room becomes occupied.[1]

2.7 LOTOS: an Event-based Language

In an event-based specification, concurrent processes communicate through events. Some languages, like Estelle, directly support both synchronised and asynchronised communication. LOTOS [103] is a synchronous language influenced initially by the Calculus of Communicating Systems (CCS) [107] and later by the Communicating Sequential Processes (CSP) [78].

Figure 2.5 on page 35 shows two processes, ***Producer*** and ***Consumer***, trying to communicate through a third process acting as a channel. The corresponding LOTOS specification is in Figure 2.6 which simulates asynchronous communication with the synchronous mechanism on four gates $pc1$, $pc2$, $cc1$ and $cc2$. Process ***Channel*** is defined with choice operator []. After synchronising on gate $pc1$, ***Channel*** can synchronise either on gate $pc2$ with ***Producer***, or on gate $cc1$ with ***Consumer***.

Formal Description Techniques (FDT) are formal methods used mainly to describe distributed systems such as communication protocols. They capture distributed behaviour with timing and state transitions [30]. Many event-based specification languages such as LOTOS, SDL and Estelle are examples of FDT.

[1] For typographical reasons, on the last line of the VDM specification in Figure 2.4, we use the left arrow symbol ← above the variable *rooms* to denote its value prior to the operation when the correct symbol is a left harpoon ↼.

2.8 Larch: an Algebraic Language

An algebraic specification defines the functions of a software application with equations. The expression on one side of an equation can be evaluated using the expression on the other side. Data types are often called *sorts* to emphasise the abstract nature of the specification languages in comparison to programming languages. Examples of algebraic specification languages are Extended ML (EML) [93], CafeOBJ [44] and Larch [65]. EML extends ML with axioms which may or may not be executable. It should not be mistaken for Extensible ML or eXtensible Markup Language (XML).

We demonstrate algebraic specification with Larch which is made up of two tiers. The interface tier, Larch Interface Language (LIL), provides the information needed to understand and use a module interface. LIL is actually a family of specification languages and there is one specification language for each supported programming language. Figure 2.7 has a LIL specification fragment for the C programming language [65, Chapter 3]. It describes a procedure that selects a task from a task queue.

The **use** statement says that the procedure uses a *TaskQueue* which we will define next. The fourth line states that the procedure *getTask* takes a queue as the parameter. Due to the asterisk symbol *, the function returns a pointer to a task not a task. In C programming practice, a function should return a pointer to the structure not the structure itself. A trailing caret ˆ denotes the value before the operation; a trailing prime $'$ denotes the value after. Keyword **result** refers to the value returned by the procedure. The *if-statement* means that if the queue is empty before the operation, no task is returned and the queue is unchanged. Otherwise the first task will be removed from the queue and be returned as the result.

Figure 2.8 on page 37 shows the counterpart of Figure 2.7 in the other tier of Larch called the shared tier where the Larch Shared Language (LSL) is used. LSL is independent of the target programming language.

A *trait* is the basic unit of specification in LSL. Keyword **introduces** defines the signatures of six operators: *new*, $\dashv$, *isEmpty*, *hasImportant*, *first* and *tail*. The *new* operator returns a *queue*. The $\dashv$ operator takes a *task* and a *queue* as parameters to return a *queue*. The remaining four operators all take a *queue* parameter. The *isEmpty* and *hasImportant* operators return a Boolean value. The *first* and *tail* operators return a *task* and a *queue* respectively.

Following keyword **asserts** are some predicates. The first predicate uses keywords **generated by** to state that all values of sort *queue* can be built from operators *new* and $\dashv$. Next is a universally quantified predicate that contains six conjuncts. Semi-colons are used to separate the conjuncts in place of the usual symbol of $\wedge$. The first conjunct states that a new queue is empty. The second conjunct states that a queue constructed with the $\dashv$ operator is not empty. The third conjunct states that a new queue does not contain any important task. The last three conjuncts define operators *hasImportant*, *first* and *tail* recursively. According to its definition, operator *first* returns important tasks before unimportant ones regardless of their positions in the queue.

2.9 SCR: a Tabular Language

The Software Cost Reduction (SCR) is a tabular notation developed by the Naval Research Laboratory [73]. It is a formal methodology for specifying and analysing real-time control systems [99]. The target users are engineers. A number of industrial organisations, including Bell Lab [77] and Ontario Hydro [113], have adapted the notation. CoRE is a version of the SCR method. It was used to document the requirements of Lockheed's C-130J Operational Flight Program which was implemented with over 100K lines of Ada code [50]. The SCR's scalability is proven. It has been applied successfully to nuclear plant systems [142], avionics systems [4] and military systems [75].

The SCR notation specifies event-driven systems as state machines [36]. To keep track of the states, SCR uses two types of auxiliary variables: *terms* and *mode classes*. Terms capture intermediate computation results. Mode classes hold values to indicate the current mode. An SCR model can have multiple mode classes.

We illustrate the SCR with a safety injection control system presented by Heitmeyer, Jeffords and Labaw [73] which is a simplified version of a specification described by Courtois and Parnas [40]. The system has three input variables, also called monitored variables. They are *WaterPres*, *Block* and *Reset*. The system has one output variable, also known as controlled variable. It is *SafetyInjection*. The values of the three monitored variables are fed to the system through three sensors. Under specific conditions, the system updates the controlled variable to inject coolant.

Table 2.2 is the Mode Transition Table for mode class *Pressure*. Its mode changes on specific events. An event, denoted by @T(...), occurs when an input variable reaches the threshold specified in the brackets.

Old Mode	**Event**	**New Mode**
TooLow	@T(WaterPres $\geq$ Low)	Permitted
Permitted	@T(WaterPres $\geq$ Permit)	High
Permitted	@T(WaterPres $<$ Low)	TooLow
High	@T(WaterPres $<$ Permit)	Permitted

Tab. 2.2: A Mode Transition Table in SCR

Mode class *Pressure* may be in one of three modes: *TooLow*, *Permitted* and *High*. The first of the four rows on the table describes that when mode class *Pressure* is *TooLow*, the event of *WaterPres* being greater than or equal to *Low* will change the mode to *Permitted*. In the centre column, *Low* and *Permit* are two constant values that concern variable *WaterPres*.

Table 2.3 is the Event Table for term *Overriden*. Under the column heading, the first two rows specify the situations when term *Overriden* will change to the value on the bottom row.

Mode	**Events**	
High	False	@T(Inmode)
TooLow, Permitted	@T(Block = On) WHEN Reset=Off	@T(Inmode) OR @T(Reset=On)
Overriden	True	False

Tab. 2.3: An Event Table in SCR

Mode	**Conditions**	
High, Permitted	True	False
TooLow	Overriden	NOT Overriden
SafetyInjection	Off	On

Tab. 2.4: A Condition Table in SCR

On the first row, the value of *False* in the second column states that under no suituations, the mode *High* (first column on the first row) can give rise to value *True* (second column on the bottom row) for term *Overriden*.

Still on the first row, the value of *@T(Inmode)* in the third column states that when mode class *Pressure* becomes *High* (first column on the first row), *Overriden* becomes *False* (third column on the bottom row).

Table 2.4 is the Condition Table for output variable *SafetyInjection*. The first row means that when the mode is *High* or *Permitted*, the value of *SafetyInjection* will be off. The second row means that when the mode is *TooLow* and *Overriden* is *True*, *SafetyInjection* will be *off*. When the mode is *TooLow* and *Overriden* is *False*, *SafetyInjection* will be *on*.

It was claimed that engineers find SCR's tabular specifications easy to create and understand [73]. Decision tables are not uncommon in other methodologies. A distinguished feature of the SCR is the specification of each output, term and mode class in a separate table. A large decision table in other notations is decomposed into small manageable ones in SCR. SCRTool is an industrial-strength toolset [72][74]. SC(R)^3 builds on top of SCRTool to emphasise requirements reuse [36].

An SCR scenario is a sequence of input variable name and value pairs. The SCR modelling tool may be used to run scenarios to display values of output variables which will be inspected by a specifier for correctness. Assertions may be placed between states to ensure that they hold during the execution of the scenarios.

In the literature, systems specified in the SCR are control systems. If the end product is not a state machine, we do not see how the use of tables can help. Additionally the SCR was designed to be used by engineers. The strong mathematical background possessed by engineers does not match the profile of a typical customer. The SCR is limited in the target application domains and users. Our scenario-driven approach aims to handle general computation usable by ordinary customers.

2.10 Alloy: an Executable Specification Language

While languages like Z and VDM can capture formal specifications, they have limited utility in automated analysis. A number of languages have been designed to address that need. They include PAISLey [147], Aslan [47], Nitpick [81] and more recently Alloy [82].

Aslantest is an interactive tool to execute formal specifications written in Aslan which is a state-based language in first-order predicate calculus. However there may be too many values to test exhaustively. To reduce the number of cases to consider, Aslantest would prompt the tester to enter a value of true or false for expressions that it cannnot evaluate easily.

The design of Alloy starts by selecting features from Z that are essential for object modelling [83]. It supports two kinds of analysis: *simulation* to generate a state or transition and *checking* to generate a counterexample which may exist. Since Z was not designed with automated analysis in mind, Alloy has to adjust its syntax and semantics to ease implementation of the analysis. Features for testers to communicate with analysis tools are provided.

For specifications expressed in the Modechart language, the Modechart Toolset can be used with the results visually shown [27] [136]. Modechart is a graphical specification language similar to Harel's Statecharts [68].

The intended usage of these executable languages is to test a generalised formal specification with examples. On the contrary, the concrete scenario approach intends to use scenarios to assist the generalisation process in the creation of formal specifications.

2.11 Rule-based Approaches

2.11.1 Production Systems

The terms *rule-based systems* and *production systems* are synonymous to some people. To be more precise, production systems always hold low-level rules written in programming languages. Rule-based systems, on the other hand, hold low-level production rules and/or high-level business rules. The latter rules may not be represented in an executable form.

The AI community use production systems to mimic human problem solving. Brownston et. al. suggest that a production system is excellent for carrying out requirements analysis for ill-defined or difficult-to-express problem domains [29, page 20]. Production systems are useful prototypes to validate customer requirements [138]. A production system has the following three main components.

1. working memory or data memory

2. production memory

3. inference engine or rule engine

The working memory holds the state of computation to be updated in program execution.

The production memory holds the production rules. A rule consists of a condition and an action. A rule condition describes the data in working memory required to fire the rule action. A rule action may update data and produce output. There is neither a sequential execution order between production rules nor explicit branching instruction.

The inference engine chooses the next rule to execute. When a rule's condition is satisfied by the data in the working memory, the rule is said to be enabled. The engine executes enabled rules until no rule conditions are satisfied. When multiple rules are enabled at the same time, the inference engine applies a conflict resolution strategy to choose one rule to execute. A popular default strategy selects the rule that references the most recently updated data.

Charles Forgy invented the Rete algorithm to power inference engines. It efficiently finds the enabled rules even when the number of rules is huge. He is responsible for a family of production systems OPS (Official Production System), OPS2, OPS5, OPS83 and OPSJ. NASA's CLIPS (C Language Integrated Production System) also uses the Rete algorithm. It was made available to the public in 1986 with fine documentation and implementations on both Windows and Mac.

For both OPS5 and CLIPS, the production rules cohabit in a single tier. If production rules are divided into two tiers, presentation and business logic, change in one tier will not affect the other tier as long as their interface is intact. Java-based production systems, for example Jess and JBoss Rule, use the two-tier approach. An outstanding issue is that rules developed for a particular production system cannot run on another platform without modifications. There are two initiatives to deal with the differences in the production system platforms. The Java Community Process (JCP) created the JSR 94 which defines the application program interface (API) to be used in the presentation tier. Another initiative RuleML works on standardising the rule language used to express the business logic tier.

2.11.2 Business-Level Rules

Business rules extend the usefulness of production rules beyond software development. They describe the business processes of organisations. To ensure consistent adherence throughout an organisation, all its business rules must be kept in a repository [76]. Ronald Ross lists some problems that the business rules approach addresses [121].

- **Ad hoc rules** inconsistently made up by employees
- **Miscommunication** among people
- **Inaccessible rules** hidden in programs, documents and records
- **Undistinguished products** due to the inability to customise them
- **Rapid changes** in the market
- **Disappearing knowledge** caused by departing employees

At the business level, a rule can be written in a natural language. Here is a business rule of a university [12]. Special terms to be defined in a glossary are underlined.

> A student can sit for an exam three times at the most in one academic year.

To implement the rule in a program, the rule may be rewritten as follows.

```
IF NumberOfTries(studentID, examID, year) < 3
THEN Register(studentID, examID)
```

An event may be required to trigger a rule. Rule representation ECA stands for event, condition and action [31]. Other rule representations are decision table, decision tree and flow diagram.

TEMPORA was an ESPRIT project supporting the complete software development cycle [106]. High-level business rules are captured and then refined to low-level rules for implementation. Blaze Advisor by Fair Isaac and JRules by ILOG are two commercial products that provide a suite of tools for the entire software life cycle.

2.11.3 Abstract State Machine (ASM)

The Abstract State Machine (ASM) was originally known as the Evolving Algebras. It is an attempt by Yuri Gurevich to bridge the gap between formal models of computation and practical specification methods [62]. The thesis is that any algorithm can be modelled at its natural abstraction level by an appropriate ASM.

The state of computation is captured in functions defined by the specifier. The basic operations of ASMs are expressed as rules in the following form.

if cond **then** Updates

where *cond* is a boolean expression and *Updates* are a finite number of function updates as follows.

$$f(t_1, \ldots, t_n) := t_0$$

In each step of computation, the conditions of all rules are evaluated. The rules with true condition are enabled and fired together to update the functions. For example, after the update, function f above remains the same as before except that it is overriden with t_0 at the arguments of $t_1, \ldots, t_n$. However if two function updates try to override a function at the same arguments with different values, no updates will take place [63, Section 2.3]. In addition to the *if-rule* above, there are also *choose-rule*, *forall-rule* and so on [66, Chapter 6].

Consider the following informal requirement.

> Whenever the cabin pressure exceeds the limits, the system shall set the Cabin Pressure Alarm to TRUE, send a warning message to the earth-bound controller that is controlling the CPM, and switch to emergency state.

Gervasi translates it to an ASM rule at the same abstraction level as the application domain [57].

```
if Exceeds( Pressure( Cabin), Limits) then
  PressureAlarm( Cabin) := TRUE
  SEND( WarningMessage, Controlling( CPM))
  State( self) := Emergency

where

SEND($msg$, $dst$) $\widehat{=}$ Channel($self$, $dst$) := Channel($self$, $dst$) $\uplus$ $\{msg\}$
```

The above rule is not exactly customer-friendly. Yet, it improves on the mainstream formal notations in understandability.

Table 2.5 on the next page shows four increasingly refined ASM rules for the same requirement. The specifier can choose the right level of abstraction to suit the stage of development.

ASM is applicable to sequential, parallel and distributed systems [64] [19] [148]. It can capture design decisions as well as requirements. A **ground model** is an ASM without any design decisions. Börger describes a 3-step

$\underbrace{\textit{the cabin pressure exceeds the limits}}_{\textit{EVENT}}$	CabinPressureExceedsLimits()
$\underbrace{\textit{the cabin pressure}}_{\textit{INFORMATION}}\ \underbrace{\textit{exceeds the limits}}_{\textit{UNARY OP}}$	ExceedsLimits(CabinPressure)
$\underbrace{\textit{the cabin pressure}}_{\textit{INFORMATION}}\ \underbrace{\textit{exceeds}}_{\textit{BINARY OP}}\ \underbrace{\textit{the limits}}_{\textit{INFORMATION}}$	Exceeds(CabinPressure, Limits)
$\underbrace{\textit{the cabin}}_{\textit{ENTITY}}\ \underbrace{\textit{pressure}}_{\textit{ATTRIBUTE}}\ \underbrace{\textit{exceeds}}_{\textit{BINARY OP}}\ \underbrace{\textit{the limits}}_{\textit{INFORMATION}}$	Exceeds(Pressure(Cabin), Limits)

Tab. 2.5: Increasingly refined ASM rules

iterative approach of building ASM ground models [24]. The first step is to *collect* the informally presented requirements and to create from them a rigorous description. The second step is to *structure* the description with parameterisation and abstraction to make its structure more transparent. The third step is to *complete* the description with more detailed customer requirements, for example boundary conditions and exception handling. The above table roughly corresponds to the first two steps of the approach.

There are a number of tools available to execute ASM specifications. AsmGofer is an ASM interpreter embedded in a functional programming language Gofer which is a subset of Haskell [125]. AsmGofer has been used to specify an executable version of a light control system [24] and a Java Virtual Machine (JVM) [134]. AsmL is another executable language created by Foundations of Software Engineering (FSE) group of Microsoft Research to work with other .NET languages [111]. Additional ASM tools include XASM, CoreASM, TASM and ASMETA.

Many rule-based approaches have been proposed. In the 70's, there were Dijkstra's guarded commands [46] and the production systems. In the 80's, there were Chandy and Misra's UNITY [35], Gurevich's ASM, and Back and Kurki-Suonio's Action System [9]. Most approaches represent rules at the programming level. ASM distinguishes itself by using the terms and concepts directly from the application domain. Communication is improved by starting from a level of abstraction more accessible to customers. The ASM method does not encompass the complete software process [80]. Its contributions begin in the requirements specification. Customers must know the rules before the rules can be encoded in ASM. On the other hand, concrete scenarios bring out the rules from customer-provided examples. Contributions of concrete scenarios begin in the requirements elicitation which drive other development activities including the requirements elicitation.

2.12 Hybrid Approaches

2.12.1 DFD + VDM

Elicitation and analysis are two distinct requirements activities. Fraser et. al. recognise that a customer-friendly language is needed during elicitation to encourage customer participation. A formal language is needed during analysis to support proofs and consistency checks. They choose data flow diagram for requirements elicitation and VDM for requirements analysis [55]. On that basis, they attempted two alternative approaches to create VDM specifications.

In the first approach, DFD's guide the manual development of VDM specifications. The input/output data flow of a process in the DFD is used to define the signature of an operation in VDM. Details such as the pre- and post-conditions are manually added to the VDM specification.

In the second approach, DFD's are used to create VDM specification semi-automatically. Given specific control structures in a DFD, for example a while-loop, the specifier can augment the DFD with a manually created decision table for automatic generation of a VDM specification.

2.12.2 Structured-Object-based-Formal Language (SOFL)

SOFL integrates structured analysis, object-orientation and formal method in a specification. It starts with a variant of DFD called *condition data flow diagram* (CDFD). OO details are then added. The formal part of the specification is written in VDM [102]. Unlike the work of Fraser *et al.*, the VDM part in SOFL only describes partial constraints. This formal part does not fully specify the program being built. It is a supplement to the specification in CDFD and OO. Software quality is assured mainly by reviews, inspections and testing. Formal proofs are optional.

2.12.3 SSADM + Z = SAZ

SAZ is another method that integrates a structured systems analysis method (SSADM) and a formal notation (Z) [104]. Based on the waterfall model, SSADM employs approachable diagrammatic and textual forms for tasks in the following modules.

1. feasibility study

2. requirements analysis

3. requirements specification

4. logical system specification

5. physical design

The three key techniques used in SSADM are **logical data modelling** for the staic aspect of data, **data flow modelling** for the dynamic aspect of data and **entity event modelling** to capture the relationship of business events and their influence on data entities. The use of the formal notation Z encourages developers to address the requirements that are not described detailed enough in the diagrams or precise enough in the texts. After removing the SSADM elements, the SAZ tutorials in [116] and [117] may appear similar to the approach we are proposing in the dissertation. A key omission in SAZ however is the use of z-scenarios, a form of examples expressed in Z notation.

2.12.4 Framework for Integrated Test (Fit)

Fit is a tool to enhance collaboration among customers, programmers and testers in software development [108]. Customers create tables documenting sample computation in HTML files. Many tools, including word processing programs, allow people without knowledge of HTML to edit HTML files. A row on a table captures the input and output values of a computation example. Programmers write small programs known as *fixtures* which check the examples on the tables. The customer, programmer or tester can run a fixture against a table. A similar table is returned with unexpected output values highlighted. Figure 2.9 shows the result of a payroll calculation [41]. New tests or examples can be easily added to a table for a rerun.

The task computes the pay based on the number of standard hours, holiday hours and the hourly rate. There are three examples in the generated HTML file. The third example is highlighted pink to indicate a difference in the expected pay of $1360 and the calculated pay of $1040. When this happens, the programmer will update the fixture until all the results are correct.

```
types
  $RoomNumber = \{1, \ldots, 100\};$
  $RoomStatus = Available \mid Occupied$
state $Reservation$ of
  $rooms : RoomNumber \xrightarrow{m} RoomStatus$
  init mk-$Reservation(rooms) \triangleq \forall\ rn \in \mathbf{dom}\ rooms \bullet rooms(rn) = Available$
end
operations
  $book\text{-}room\ (roomno : RoomNumber)$
  ext wr $rooms : RoomNumber \xrightarrow{m} RoomStatus$
  pre $roomno \in \mathbf{dom}\ rooms \land rooms(roomno) = Available$
  post
    let $st : RoomStatus = Occupied$ in
      $rooms = \overleftarrow{rooms} \dagger \{roomno \mapsto st\};$
```

Fig. 2.4: A VDM specification

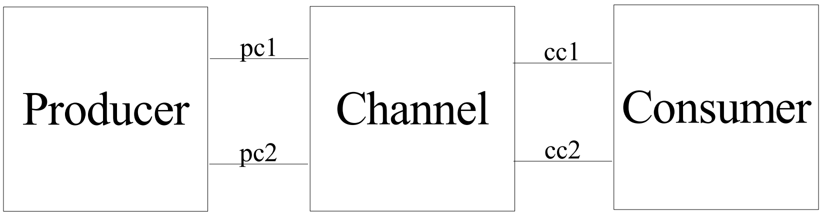


Fig. 2.5: Two processes communicating over a channel process

```
process Producer [pc1, pc2] : exit :=
   pc1; pc2; exit
endproc

process Consumer [cc1, cc2] : exit :=
   cc1; cc2; exit
endproc

process Channel pc1, pc2, [cc1, cc2] : exit :=
   pc1;
   (
      pc2; cc1; cc2; exit
   []
      cc1; pc2; cc2; exit
   )
endproc
```

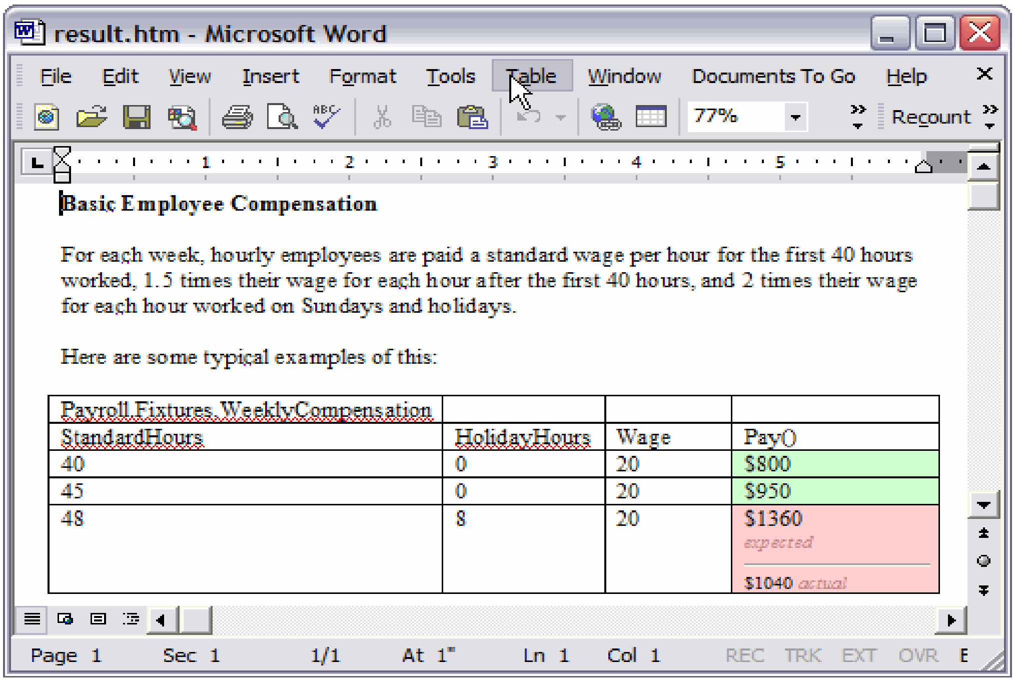
Fig. 2.6: A LOTOS specification

```
uses TaskQueue;
mutable type queue;
immutable type task;
task *getTask(queue q) {
    modifies q;
    ensures
        if isEmpty(qˆ)
            then result = NIL ∧ unchanged(q)
            else (*result)′ = first(qˆ) ∧ q′ = tail(qˆ); }
```

Fig. 2.7: An LIL specification for the C language

```
TaskQueue: trait
   includes Nat
   task tuple of id: Nat, important: Bool
   introduces
      new: → queue
      __ ⊣ __ : task, queue → queue
      isEmpty, hasImportant: queue → Bool
      first: queue → task
      tail: queue → queue
   asserts
      queue generated by new, ⊣
      ∀ t: task, q: queue
         isEmpty(new);
         ¬ isEmpty(t ⊣ q);
         ¬ hasImportant(new);
         hasImportant(t ⊣ q) == t.important ∨ hasImportant(q);
         first(t ⊣ q) == if t.important ∨ ¬ hasImportant(q)
                          then t
                          else first(q);
         tail(t ⊣ q) == if first(t ⊣ q) = t
                          then q
                          else t ⊣ tail(q)
```

Fig. 2.8: An LSL specification corresponding to Figure 2.7



Basic Employee Compensation

For each week, hourly employees are paid a standard wage per hour for the first 40 hours worked, 1.5 times their wage for each hour after the first 40 hours, and 2 times their wage for each hour worked on Sundays and holidays.

Here are some typical examples of this:

Payroll.Fixtures.WeeklyCompensation			
StandardHours	HolidayHours	Wage	Pay()
40	0	20	$800
45	0	20	$950
48	8	20	$1360 expected $1040 actual

Fig. 2.9: An HTML file returned by Fit

2.12.5 Controlled Natural Languages (CNLs)

A controlled natural language (CNL) is a subset of a natural language (NL) with a well-defined lexicon, syntax and semantics. Attempto [128] and PENG (Processable ENGlish) [127] are two controlled natural languages. Sentences are written with predefined and user-defined words. Predefined words are *if*, *then*, *and*, *or*, *not*, *after*, *while*, *each*, *the*, *is* and so on. User-defined words express concepts in a specific application domain. Here is an example [56].

> If a passenger alerts a driver of a train
> then the driver stops the train in a station.

Writing correct sentences according to the restrictions can be a slow and challenging activity. A look-ahead editor ECOLE lowers the learning curve and increases specifiers' productivity [129]. It gives syntactic hints to the specifiers as they type. It also displays the paraphrase of a completed sentence for the specifier to confirm the intended meaning. A tool can translate a sentence into a discourse representation structure (DRS) which is a syntactical variant of first-order predicate logic. For example, the earlier sentence would have the following DRS.

```
IF
        [A, B, C, D]
        passenger(A)
        driver(B)
        train(C)
        of(B,C)
        event(D, alert(A,B))
THEN
        [E, F]
        station(E)
        event(F, stop(B,C))
        location(F, in(E))
```

The translated DRS's have been converted to executable languages, for example, Attempto to Prolog and PENG to OTTER [94]. Inferences can be made by relating the sentences. For example, suppose we have the following sentence.

> A passenger alerts a driver of a train.

This simple sentence can be combined with the earlier *if-sentence* to give the following.

> The driver stops the train in a station.

The sentences in a controlled natural language can express business rules formally.

2.13 Temporal Logic

In temporal logic, the truth values of propositions depend on time [16] [97]. Consider the following statements A and B.

A. The sun is rising.

B. The sun is setting.

We can write A *leads-to* B to mean that ”Sunrise leads to sunset”. In addition to the *leads-to* operator, there are other operators such as *always*, *eventually* and *until*. Temporal logic can be used to describe requirements of concurrent programs, for example liveness and safety properties. Though concurrency is not our emphasis, temporal logic is too important a topic unmentioned.

As operators are added, a temporal logic expression quickly becomes difficult to read. Message sequence charts [69] and their descendants UML sequence diagrams can express event ordering as shown in Figure 2.2 on page 17. But they have a few shortcomings. It is unclear whether the later events in a diagram are mandatory or optional. There is no constuct in them to concisely express that two events may reverse order or an event can substitute another event. Property Sequence Chart (PSC) is a visual language designed to overcome the above-mentioned deficiency by striking a balance between the friendlness of sequence diagrams and the expressiveness of temporal logic [8].

PSC expresses requirements in terms of event ordering while concrete scenario approach uses state changes with examples. In principle, events may be parametised to express additional details found in concrete scenarios. In practice, parametised events are seldom used likely due to its awkwardness.

2.14 Goal-oriented Requirements Specifications

Requirements can be written at different levels of abstraction. Svetinovic has identified five levels of requirements from high to low [137]. He commends the recent emphasis by researchers on the higher-level requirements which better serve the customers in their business goals.

1. **Business-level** requirements concern the business goals to be fulfilled by the system.

2. **Domain-level** requirements concern user goals and user tasks.

3. **Product-level** requirements involve the specification of functional lists, use cases, data input and data output.

4. **Design-level** requirements, commonly expressed in UML, serve as the transition from the product-level specification to the code-level specification.

5. **Code-level** requirements, often expressed as pseudo-code, are a part of the programming actitivty.

The approaches described earlier in the chapter mainly deal with the lower levels in Svetinovic's classification. In the past 10 to 15 years, we have seen a noticeable increase in the attention to the top two levels of requirements. The attention to business and user goals is well justified lest the resulting software ends up doing a nice job for inappropriate goals. Problem Frames [85] [86], KAOS and *i** are three of the notations that emphasize high-level goals.

2.14.1 KAOS

KAOS is a goal-oriented requirements specification language which describes the relationships between objects (active or passive), operations and goals [141]. Following is a goal specification in KAOS by Brandozzi and Perry [26]. The goal here is to maintain the confidentiality of submissions in the review process of a publication. *DocumentCopy*, *Knows*, *People* are data components. The goal of *ConfidentialityOfSubmissions* is refined to the subgoals of *ConfidentialityOfSubmissionDocument* and *ConfidentialityOfIndirectSubmission*. At the end is an informal definition of the goal.

```
Goal Maintain[ConfidentialityOfSubmissions]
InstanceOf SecurityGoal
Concerns DocumentCopy, Knows, People
ReducedTo
    ConfidentialityOfSubmissionDocument
    ConfidentialityOfIndirectSubmission
InformalDef A submission must remain confidential.
            A paper that has to be submitted has
            to remain confidential.
```

The ultimate goal of the publisher is to publish quality articles. This fundamentalgoal can be refined to a number of subgoals including the following.

```
Maintain[QualityOfEditorialDecisions]
Achieve[EnoughQuantityOfPublishedArticles]
Avoid[ConflictOfInterestsWithAssociatedEditor]
```

*2.14.2 i**

In an organization, heterogenous actors may have competing and interdependent goals. Actors will have to undertake their tasks to accomplish these goals. *i** attempts to model this kind of **distributed intentionality** [1].

*i** framework uses two kinds of models. A Strategic Dependency (SD) model shows the dependency of an actor on the work of another actor. Actors are shown in circles. Figure 2.10, taken from the PowerPoint used by Yu for [146], has three actors: car owner, body shop and insurance company. They are dependent on each other in multiple ways. In the left side of the figure, the car owner depends on the body shop to have the car repaired. The large oval between the two actors indicates that this is a **goal dependency**. The tiny crescent shapes on both sides of the oval shows the direction of the dependency. The rectangle labeled "Premium payment"near the top of the figure is a **resource dependency**. A soft-goal is a less precise goal. Therefore a **soft-goal dependency** is denoted by a cloud shape, for example the third dependency from the top "Customer Be Happy". Finally, there is a **task dependency** not shown in the figure. For example, an insurance company depends on an appraiser for the task to appraise damages.

A Strategic Rationale (SR) model is built on top of an SD model by adding details of goals, soft goals, tasks and resources. Two kinds of important details are shown with means-ends links and task-decomposition links. A **means-end link** shows the means of a task attaining the end of a goal. A **task-decomposition link** shows a task being decomposed into subgoals, subtasks, resources or softgoals. Figure 2.11, also by Yu the creator of *i**, shows in its top right corner that the task to handle a claim attains the goal of claim settlement. The task to handle a claim is decomposed into three subtasks to verify the policy, prepare a settlement offer and make an offer to settle.

The Strategic Dependency Model

auto insurance – example 3
"Let the Body Shop handle it."

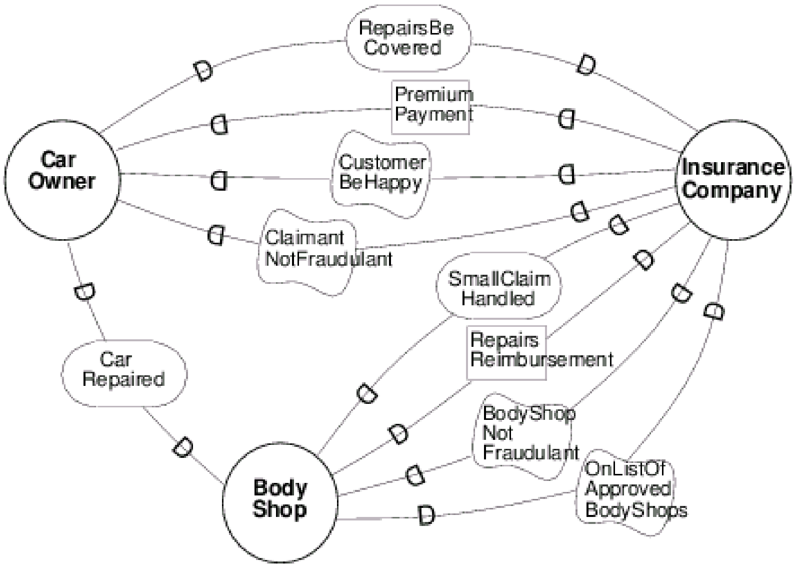


Fig. 2.10: A strategic dependency model for car insurance

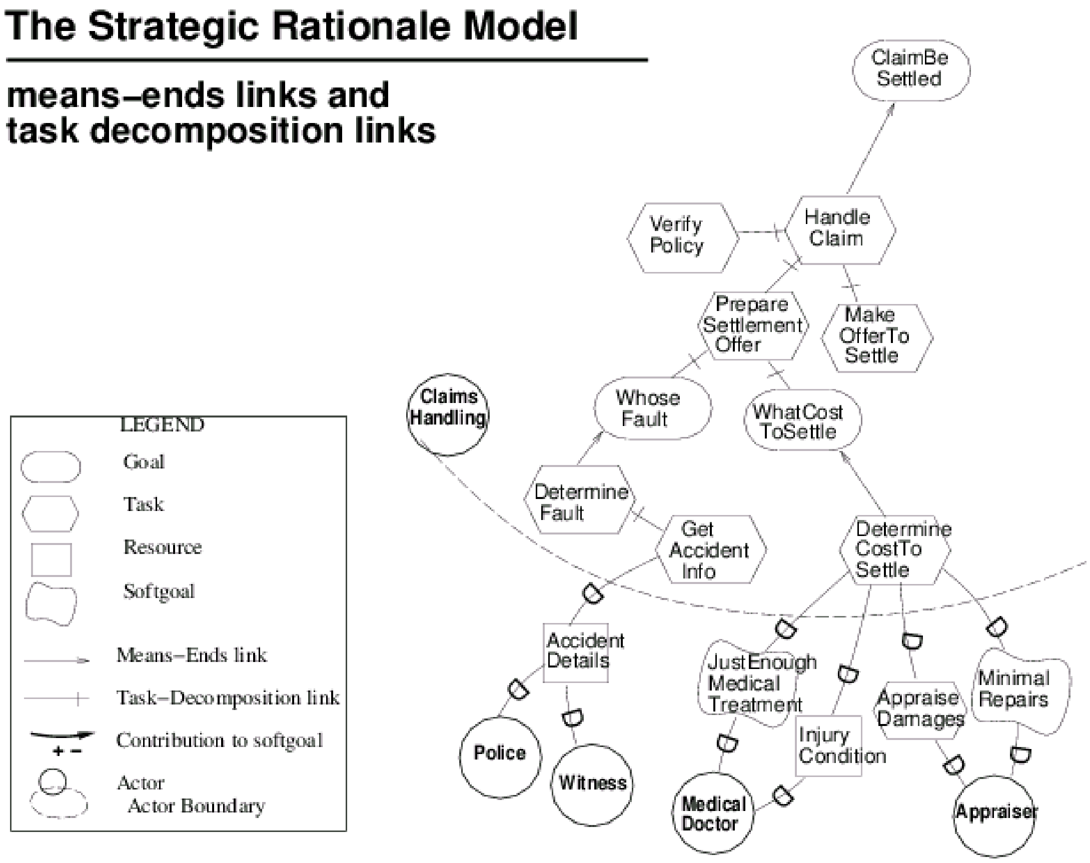


Fig. 2.11: A strategic rationale model for claims handling

2.14.3 User Requirements Notation (URN)

User Requirements Notation (URN) is an international standard of the International Telecommunications Union (ITU). It is the work of ITU's Telecommunication Standardization Sector (ITU-T). The semi-formal, lightweight graphical language models and analyzes requirements in the form of goals and scenarios [6]. URN consists of two complementary notations: Goal-oriented Requirement Language (GRL) and Use Case Maps (UCM).

GRL models goals and other intentional concepts. It is mainly used for the expression and reasoning of non-functional requirements, quality attributes, rationales, alternatives and tradeoffs. It is based on $i*$ and the NFR framework [37] where NFR stands for non-functional requirements.

UCM is used for modelling scenario concepts covering functional requirements, operational constraints, performance and architectural reasoning. UCM is a kind of flowcharts relating agents, processes and components with fork and join flows. An and-fork splits the flow to two parallel paths. An or-fork chooses a flow from two possible paths.

2.14.4 From Requirements to Architecture

Brandozzi and Perry developed the *Preskriptor* process that can be used to derive an architecture from goals expressed in KAOS. The process involves assigning architectural components to satisfy the goals. There are processing components, data components and connector components. A component can be defined in terms of simpler components. Figure 7 in [26] has an example of a Preskriptor specification which describes an architecture to meet the goals.

Similarly, *Tropos* is a formal language with an underlying methodology that turns $i*$ models into architectural and detailed design [34].

STRAW'01 and STRAW'03 published a few approaches that derive architectures from requirements [110][48][60][101][130]. These approaches generally presume a top-down software development methodology which takes us from requirements to architecture, then from architecture to detailed design. Nuseibeh's approach is unique among other approaches in the two conferences in the use of the *Twin Peaks* development model. It is a partial and simplified version of the spiral model [21] where details of the requirements are added side-by-side with the details in the architecture. This makes Nuseibeh's approach more compatible with agile development methods such as XP. We will comeback to the discussion of top-down versus bottom-up in the concluding chapter and how concrete scenarios fit into the big picture.

2.15 Chapter Conclusion

A formal specification fares well in four of the nine desirable qualities listed in Chapter 1. It allows reliable determination of correctness and consistency. It is as unambiguous and verifiable as it can get. For the qualities of completeness, ranking for importance, modifiability and traceability, it is dependent on the individual formal language. Its main weakness is the lack of understandability by customers. Unfortunately, customer involvement is the most important success factor [61]. Adding to it the increased costs and time-to-market [132, page 193], the lack of impact by formal specification languages on the industry can be explained [59].

Two formal approaches make significant strides in understandability. The ASM approach allows specifications to be written with terms taken directly from application domains though customers may need to get used to the syntax of procedural calls and parameters. The CNL approach takes it a step further by allowing requirements to be written as precise complete sentences in natural languages. Both ASM and CNL approaches capture requirements as general rules with variables. Concrete scenarios can supplement both with examples written with actual data. Concrete scenarios document specific examples which are generalised to rules or operation schemas as shown in this thesis. Without concrete scenarios, customers may have trouble generalising the examples to rules in their heads. Concrete scenarios can be used as documentation to improve communications. Without scenarios to link their actions, rules are independent entities. Scenarios put them into perspective. They improve requirements understanding and reduce the likelihood of unused rules. Concrete scenarios guide formal specification writing and double as test cases for verification.

Concrete scenarios only deal with low-level and at best mid-level functional requirements. To manage high-level business goals and non-functional requirements, a goal-oriented requirements specification presented in the previous section is in order.

3. CUSTOMER-FRIENDLY CONCRETE SCENARIOS

A small experiement involved 62 computer science or software engineering students [51]. Most of them had a basic training in discrete mathematics. Their training in Z notation ranged from a 3-hour lecture to a full semester. They were asked to answer three questions about a simple Z specification with less than 20 lines. The students were allowed to take however long they needed. Most had spent somewhere between 200 to 800 seconds. Finney and Fedoree came to their observation and conclusion as follows [52].

> As it was clear that 19 of the (62) subjects did not understand the specification sufficiently to answer any of the questions correctly despite their background, then we should not expect clients and software engineers to master Z and other formal methods without training.

If customers do not understand a formal specification, they cannot give feedback on its correctness or completeness. To address this problem, we advocate an approach in which computation steps are described by their effects on the system state expressed in actual data rather than variables. The customer uses simple English to describe a step by its input, output and system states. The specifier translates the descriptions to a formal notation using very few simple concepts. The approach allows customers to take part in the creation of formal specifications. The odds of formal specifications fully capturing customer requirements are improved.

The objective of this chapter is to introduce a scenario-driven approach for specification writing with a focus on its customer-friendliness. Some of the contents in this chapter have been published previously by Au, Stone and Cooke [7]. In that publication, we called our scenarios *precise scenarios* which are renamed *concrete scenarios* in this thesis. The new name suggests the use of concrete data which is a more distinctive feature of our scenarios.

The approach is introduced by example. This example is intended to demonstrate the customer-friendliness of the approach. Other chapters deal with other aspects. In this chapter, Section 3.1 describes the application domain of our example. Section 3.2 outlines the approach and explains the preparation. Sections 3.3 to 3.6 show the derivation of schemas from scenarios. In Sections 3.7 and 3.8, we run the schemas through additional scenarios to detect under- and over-specification. Section 3.9 demonstrates how to prove an operation total. Section 3.10 justifies the claim that concrete scenarios are more understandable to customers than formal specifications. Section 3.11 is our chapter conclusion.

3.1 Warehouse Ordering Problem

The warehouse ordering problem we use is credited to Habrias and Frappier [66]. Chapter 1 of the book presents a Z specification by Bowen which is almost identical to our Z specification derived from concrete scenarios.

The customer needs an up-to-date record of the stock in the warehouse. The four operations are order creation, fulfillment, cancellation and stock replenishment. When an order is first created in the system, it is in status *pending*. After the order is filled, the status becomes *invoiced*. Only a pending order may be cancelled. The customer does not need a trail of cancelled orders.

3.2 Scenario-Driven Specification

Figure 3.1 shows the six artefacts in the scenario-driven specification writing approach that we advocate. An arrow represents the dependency between two artefacts. When an artefact is modified, its dependent artefacts should be revised. The artefacts that the specifier is responsible for may be tailored to a formalism other than the Z notation.

1. **Sentence Templates** – System states, input and output are written in simple sentences. The same sentence template is used to capture the same kind of information.

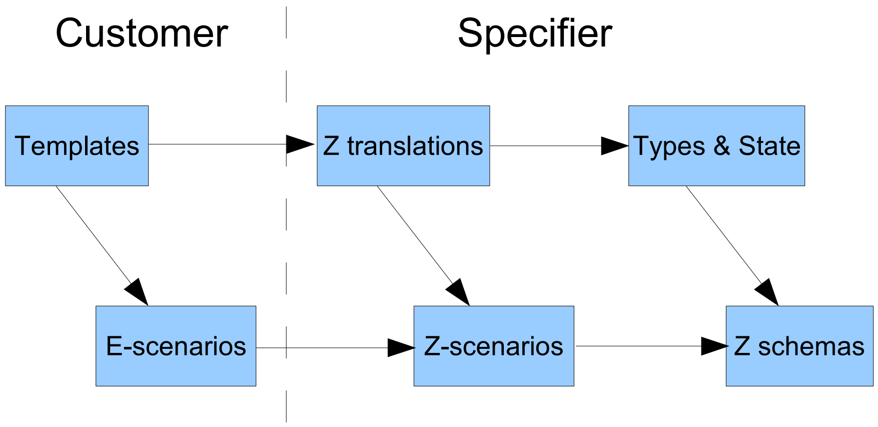


Fig. 3.1: A scenario-driven specification process

2. **E-scenarios** – A scenario is a sequence of successive steps. A step has a pre-state, post-state, input and output. The customer write them in English according to the sentence templates.

3. **English to Z Translation Rules** – The specifier finds a Z expression to suit each sentence template.

4. **Types and State** – The specifier defines Z types and system state schema.

5. **Z-scenarios** – The specifier translates E-scenarios to Z-scenarios using the rules above.

6. **Z Operation Schemas** – The specifier generalises Z-scenarios to Z operation schemas. When in doubt, the specifier consults with the customer to confirm the validity of the generalisation.

The remainder of Section 3.2 creates the first four artefacts. Sections 3.3 to 3.6 create the last two artefacts.

3.2.1 Templates in Words

Table 3.1 on the next page has the customer descriptions of sample interactions between the users and the system. Table 3.2 describes four kinds of information about the system state: stock quantity, order content, order status and free order id. By varying the underlined parts, different interactions and states can be represented using the same templates.

Interactions	**Templates**
Create Order	Create an order for 4 nuts and 4 bolts. A new order with id 3 is created.
Invoice Order	Fill and invoice order id 2.
Cancel Order	Cancel order id 2.
Enter Stock	Replenish stock with 80 nuts and 70 bolts.
Report	Operation report is okay.

Tab. 3.1: Interaction templates

States	**Templates**
Stock Quantity	There are 5 nuts in stock. There are 6 bolts in stock.
Order Content	Order 1 is for 2 nuts and 2 bolts. Order 2 is for 3 bolts.
Order Status	Order 1 is invoiced. Order 2 is pending.
Free Order Id	Id 3 is free for future use. Id 4 is free for future use.

Tab. 3.2: State templates

3.2.2 E-Scenarios

Table 3.3 on the following page has an E-scenario describing the creation of an order expressed with the sentence templates introduced earlier. We highlight the parts being deleted from the pre-state or added to the post-state by underlines. A computation step is made up of input, output, pre-state and post-state. A scenario usually has multiple steps. To give the readers a gentle introduction, all scenarios in this chapter have only one step.

The customer is responsible for the English-based artefacts: sentence templates and E-scenarios. The specifier is responsible for the remaining artefacts because they are Z-based. After this point, the customer and

Input

Create an order for 4 nuts and 4 bolts.

Pre-State

There are 5 nuts in stock. There are 6 bolts in stock.
Order 1 is for 2 nuts and 2 bolts. Order 2 is for 3 bolts.
Order 1 is invoiced. Order 2 is invoiced.
Id 3 is free for future use. Id 4 is free for future use.

Post-State

There are 5 nuts in stock. There are 6 bolts in stock.
Order 1 is for 2 nuts and 2 bolts. Order 2 is for 3 bolts.
Order 3 is for 4 nuts and 4 bolts.
Order 1 is invoiced. Order 2 is invoiced.
Order 3 is pending. Id 4 is free for future use.

Output

A new order with id 3 is created.
Operation report is okay.

Tab. 3.3: **E-scenario** *CreateOrder*

specifier can continue to communicate using English-based artefacts.

3.2.3 English to Z Translations

The specifier finds a suitable formal expression for each sentence template. For our warehouse problem, two Z notions suffice: a set enclosed in braces and an ordered pair on both sides of maplet symbol $\mapsto$. Table 3.4 shows input and output parameters denoted by trailing question marks and exclamation marks respectively. Table 3.5 on the following page shows state information held in variables. The translation rules revealed in the tables can be used to translate E-scenarios to Z-scenarios. From this point on, we will start using the Z notation which the reader may not be familiar with. We try to alleviate the discomfort that it may cost with appropriate explanations.

Interactions in English	**Z Expressions**
Create an order for 4 nuts and 4 bolts.	$order? = \{nut \mapsto 4, bolt \mapsto 4\}$
A new order with id 3 is created.	$newId! = 3$
Fill and invoice order id 2.	$id? = 2$
Cancel order id 2.	$id? = 2$
Refill stock with 80 nuts and 70 bolts.	$newStock? = \{nut \mapsto 80, bolt \mapsto 70\}$
Operation report is okay.	$report! = OK$

Tab. 3.4: Warehouse Interactions – from English to Z

States in English	**Z Expressions**
There are 5 nuts in stock. There are 6 bolts in stock.	$stock = \{nut \mapsto 5, bolt \mapsto 6\}$
Order 1 is for 2 nuts and 2 bolts. Order 2 is for 3 bolts.	$orders = \{1 \mapsto \{nut \mapsto 2, bolt \mapsto 2\},$ $2 \mapsto \{bolt \mapsto 3\}\}$
Order 1 is invoiced. Order 2 is pending.	$orderStatus = \{1 \mapsto invoiced,$ $2 \mapsto pending\}$
Id 3 is free for future use. Id 4 is free for future use.	$freeIds = \{3, 4\}$

Tab. 3.5: Warehouse States – from English to Z

3.2.4 Type Definitions and State Schema

From the Z expressions, the developer would realise that basic types are needed to identify individual orders and products. We define types *ORDERID* and *PRODUCT* with the following statement so that they can be used in the Z specification without concern of their actual implementation.

$$[ORDERID, PRODUCT]$$

A bag of *PRODUCT* is equivalent to a partial function from *PRODUCT* to the set of positive natural numbers $\mathbb{N}_1$. We define type *ORDER* for later declarations of order and stock.

$$ORDER == \{order : \mathrm{bag}\, PRODUCT \mid order \neq \emptyset\}$$

When an order is newly created, it is in status *pending*. After the order has left the warehouse, its status changes to *invoiced*. These are the only two statuses that concern us here. If we were to consider payment scenarios, we would need another status *paid*.

$$STATUS ::= pending \mid invoiced$$

Schema *OrderSystem* uses four variables to represent the system state. Their types must agree with the sample Z expressions used in Table 3.5. The symbols $\pfun$ and $\mathbb{P}$ represent partial function and power set respectively.

$$
\begin{array}{|l}
\textit{OrderSystem} \\
\hline
\textit{stock} : \mathrm{bag}\, \textit{PRODUCT} \\
\textit{orders} : \textit{ORDERID} \pfun \mathrm{bag}\, \textit{PRODUCT} \\
\textit{orderStatus} : \textit{ORDERID} \pfun \textit{STATUS} \\
\textit{freeIds} : \mathbb{P}\, \textit{ORDERID} \\
\hline
\mathrm{dom}\, \textit{orders} = \mathrm{dom}\, \textit{orderStatus} \\
\mathrm{dom}\, \textit{orders} \cap \textit{freeIds} = \emptyset \\
\hline
\end{array}
$$

After the horizontal dividing line in the schema definition, there are two invariants. The keyword **dom** stands for domain. The first invariant ensures that an order id must simultaneously exist in both *orders* and *orderStatus* or not at all. The second invariant prevents an order id from being used and at the same time available for new orders. It is common for Z experts to write invariants before operation schemas as we do here. But this is not necessary. Our argument is that we do not have a clear understanding of the application until we have considered a number of scenarios. Writing down invariants early is unrealistic for many real-world projects.

We want to know if an operation has succeeded or not. If an operation fails, we shall report the reason of the failure. For convenience, we include all the possible report values in the following definition of type *REPORT*. In practice, the values will only be discovered one by one as scenarios are considered. Their meanings will become clearer then.

$$
\begin{array}{ll}
\textit{REPORT} ::= & \textit{OK} \mid \textit{no\_more\_ids} \mid \textit{order\_not\_pending} \mid \\
& \textit{id\_not\_found} \mid \textit{not\_enough\_stock}
\end{array}
$$

3.3 Create Order

In Sections 3.3 to 3.6, we derive Z schemas from Z-scenarios. Perhaps with the help from the customer, the specifier discovers relationships of the data in state variables and in I/O parameters. The relationships are codified as predicates in schemas.

The E-scenario in Table 3.3 on page 52 is translated to the Z-scenario in Table 3.6 on the next page. The translation rules are the same as those used in Tables 3.4 and 3.5. Due to the trivial correlations between E-scenarios and Z-scenarios, we will skip the E-scenarios of other operations. The table in the Z-scenario shows the state variables in four columns. Below the rows of variable names, the first row has their pre-state values and the second row has the post-state values.

$order? = \{nut \mapsto 4, bolt \mapsto 4\}_a$

$stock$	$orders$	$orderStatus$	$freeIds$
$\{nut \mapsto 5,$ $bolt \mapsto 6\}$	$\{1 \mapsto \{nut \mapsto 2, bolt \mapsto 2\},$ $2 \mapsto \{bolt \mapsto 3\}\}$	$\{1 \mapsto invoiced,$ $2 \mapsto invoiced\}$	$\{3_b, 4\}$
...	$\{1 \mapsto \{nut \mapsto 2, bolt \mapsto 2\},$ $2 \mapsto \{bolt \mapsto 3\},$ $3_b \mapsto \{nut \mapsto 4, bolt \mapsto 4\}_a\}$	$\{1 \mapsto invoiced,$ $2 \mapsto invoiced,$ $3_b \mapsto pending\}$	$\{4\}$

$newId! = 3_b \wedge report! = OK$

Tab. 3.6: **Z-scenario** *CreateOrder*

Scenario *CreateOrder* documents a successful attempt to create an order. The I/O parameter values that appear in the state variables are subscripted, in this case $\{nut \mapsto 4, bolt \mapsto 4\}_a$ and 3_b. The corresponding data in the pre- and post-states are also subscripted to show their equality with the I/O parameters. Some I/O parameters are not subscripted, for example report value OK, because it is not directly related to any state variable. The relationships between subscripted values in the state, input and output will be captured in the predicates of a schema as follows.

CreateOrder

$$\begin{array}{l}
\Delta \mathit{OrderSystem} \\
\mathit{order}? : \mathit{ORDER} \\
\mathit{newId}! : \mathit{ORDERID} \\
\mathit{report}! : \mathit{REPORT} \\
\hline
\mathit{newId}! \in \mathit{freeIds} \\
\mathit{stock}' = \mathit{stock} \\
\mathit{orders}' = \mathit{orders} \cup \{\mathit{newId}! \mapsto \mathit{order}?\} \\
\mathit{orderStatus}' = \mathit{orderStatus} \cup \{\mathit{newId}! \mapsto \mathit{pending}\} \\
\mathit{freeIds}' = \mathit{freeIds} \setminus \{\mathit{newId}!\} \\
\mathit{report}! = \mathit{OK}
\end{array}$$

Symbol Δ in the first declaration of the schema denotes that variables in *OrderSystem* will be updated. The presence of 3_b in *freeIds* is captured by the first predicate. Its absence in $\mathit{freeIds}'$ is represented by the fifth predicate. Due to the generalisation, output variable name $\mathit{newId}!$ is used in the predicates in place of value 3_b.

The 3-dot symbol ... in the scenario is our way to express an unchanged value. The unchanged value of stock' is specified by the second predicate. In the third and fourth predicates, orders' and $\mathit{orderStatus}'$ are expressed in terms of variables rather than data values wherever possible.

Table 3.7 on the following page is a scenario of an unsuccessful attempt to create an order due to fully depleted order ids. No data in the pre-state or post-state are subscripted to match the I/O parameters because their values are irrelevant. The precondition of the scenario is simply an empty set of *freeIds*. The symbol Ξ in the first declaration of schema *NoMoreIdsError* indicates that variables in *OrderSystem* remain unchanged.

$order? = \{nut \mapsto 7\}$

stock	*orders*	*orderStatus*	*freeIds*
$\{nut \mapsto 5,$ $bolt \mapsto 6\}$	$\{1 \mapsto \{nut \mapsto 2, bolt \mapsto 2\},$ $2 \mapsto \{bolt \mapsto 3\},$ $3 \mapsto \{nut \mapsto 4, bolt \mapsto 4\},$ $4 \mapsto \{bolt \mapsto 8\}\}$	$\{1 \mapsto invoiced,$ $2 \mapsto invoiced,$ $3 \mapsto pending,$ $4 \mapsto pending\}$	$\{\ \}$
...	...	...	...

$report! = no\_more\_ids$

Tab. 3.7: **Z-scenario** *NoMoreIdsError*

NoMoreIdsError

$\Xi OrderSystem$

$order? : ORDER$

$report! : REPORT$

$freeIds = \emptyset$

$report! = no\_more\_ids$

Together the two schemas can be used to define *CreateOrderOp* that covers all situations.

$$CreateOrderOp \mathrel{\widehat{=}} CreateOrder \lor NoMoreIdsError$$

3.4 Invoice Order

Table 3.8 desribes an invoicing operation that updates stock quantities and changes the order status from *pending* to *invoiced*. Though we are not showing E-scenarios anymore, they may be created before the Z-scenarios as part of requirements elicitation.

$id? = 2_a \land \{nut \mapsto 4, bolt \mapsto 3\}_b \sqsubseteq \{nut \mapsto 5, bolt \mapsto 9\}_c$

stock	*orders*	*orderStatus*	*freeIds*
$\{nut \mapsto 5,$ $bolt \mapsto 9\}_c$	$\{1 \mapsto \{nut \mapsto 2\},$ $2_a \mapsto \{nut \mapsto 4, bolt \mapsto 3\}_b\ \}$	$\{1 \mapsto invoiced,$ $2_a \mapsto pending\}$	$\{3, 4\}$
$\{nut \mapsto 1,$ $bolt \mapsto 6\}_d$	$\ldots$	$\{1 \mapsto invoiced,$ $2_a \mapsto invoiced\}$	$\ldots$

$report! = OK \land$
$\{nut \mapsto 1, bolt \mapsto 6\}_d = \{nut \mapsto 5, bolt \mapsto 9\}_c \uminus \{nut \mapsto 4, bolt \mapsto 3\}_b$

Tab. 3.8: **Z-scenario** *InvoiceOrder*

On the third column, the status of order id 2_a is *pending* in the pre-state. Round brackets stand for function application, for example, $orderStatus(2)$ returns *pending*. After the customer has confirmed this to be a precondition of the operation, we capture it as the second predicate after replacing value 2_a with variable *id*?. Its updated value of *invoiced* in the post-state is captured in the sixth predicate using override symbol $\oplus$. Symbol $\sqsubseteq$ denotes pairwise $\leq$ comparisons between two bags. It is more convenient than comparing individual product counts in the order and in stock. This precondition is captured in the third predicate of the schema. Values are replaced by state or I/O variables, $\{nut \mapsto 5, bolt \mapsto 9\}_c$ by *stock* and $\{nut \mapsto 4, bolt \mapsto 3\}_b$ by $orders(id?)$ according to the equality stipulated by the subscripts. Symbol $\uminus$ in the postcondition denotes pairwise subtractions between two bags. After the generalisation that replaces values with variables, the postcondition becomes the fourth predicate.

$$
\begin{array}{|l}
\multicolumn{1}{l}{\mathit{InvoiceOrder}} \\
\Delta \mathit{OrderSystem} \\
\mathit{id}? : \mathit{ORDERID} \\
\mathit{report}! : \mathit{REPORT} \\
\hline
\mathit{id}? \in \operatorname{dom} \mathit{orderStatus} \\
\mathit{orderStatus}(\mathit{id}?) = \mathit{pending} \\
\mathit{orders}(\mathit{id}?) \sqsubseteq \mathit{stock} \\
\mathit{stock}' = \mathit{stock} \uplus \mathit{orders}(\mathit{id}?) \\
\mathit{orders}' = \mathit{orders} \\
\mathit{orderStatus}' = \mathit{orderStatus} \oplus \{\mathit{id}? \mapsto \mathit{invoiced}\} \\
\mathit{freeIds}' = \mathit{freeIds} \\
\mathit{report}! = \mathit{OK}
\end{array}
$$

We are showing three exceptional scenarios of this operation and their schemas. Table 3.9 describes an attempt to invoice an order that does not exist. The scenario is generalised to schema *IdNotFoundError* defined below.

$\mathit{id}? = 3$

stock	*orders*	*orderStatus*	*freeIds*
$\{\mathit{nut} \mapsto 5,$ $\mathit{bolt} \mapsto 9\}$	$\{1 \mapsto \{\mathit{nut} \mapsto 2, \mathit{bolt} \mapsto 2\},$ $2 \mapsto \{\mathit{bolt} \mapsto 3\}\}$	$\{1 \mapsto \mathit{invoiced},$ $2 \mapsto \mathit{pending}\}$	$\{3, 4\}$
...	...	...	...

$\mathit{report}! = \mathit{id\_not\_found}$

Tab. 3.9: **Z-scenario** *IdNotFoundError*

$$
\begin{array}{|l}
\multicolumn{1}{l}{\mathit{IdNotFoundError}} \\
\Xi \mathit{OrderSystem} \\
\mathit{id}? : \mathit{ORDERID} \\
\mathit{report}! : \mathit{REPORT} \\
\hline
\mathit{id}? \notin \operatorname{dom} \mathit{orderStatus} \\
\mathit{report}! = \mathit{id\_not\_found}
\end{array}
$$

Table 3.10 on the next page describes an attempt to invoice an order that has the wrong status. The scenario is generalised to schema *OrderNotPendingError* shown on next page.

$id? = 1_a$

$stock$	$orders$	$orderStatus$	$freeIds$
$\{nut \mapsto 5,$ $bolt \mapsto 9\}$	$\{1 \mapsto \{nut \mapsto 2, bolt \mapsto 2\},$ $2 \mapsto \{bolt \mapsto 3\}\}$	$\{\ 1_a \mapsto invoiced,$ $2 \mapsto pending\}$	$\{3, 4\}$
...	...	...	...

$report! = order\_not\_pending$

Tab. 3.10: **Z-scenario** *OrderNotPendingError*

OrderNotPendingError

$\Xi OrderSystem$

$id? : ORDERID$

$report! : REPORT$

$orderStatus(id?) \neq pending$

$report! = order\_not\_pending$

Table 3.11 describes an attempt to invoice an order when the warehouse does not have sufficient stock to fill the order. The scenario is generalised to schema *NotEnoughStockError*.

$id? = 2_a \wedge 77_b > 9_c$

$stock$	$orders$	$orderStatus$	$freeIds$
$\{nut \mapsto 5,$ $bolt \mapsto 9_c\}$	$\{1 \mapsto \{nut \mapsto 2, bolt \mapsto 2\},$ $2_a \mapsto \{bolt \mapsto 77_b\}\}$	$\{1 \mapsto invoiced,$ $2 \mapsto pending\}$	$\{3, 4\}$
...	...	...	...

$report! = not\_enough\_stock$

Tab. 3.11: **Z-scenario** *NotEnoughStockError*

$$
\begin{array}{|l}
\multicolumn{1}{l}{\mathit{NotEnoughStockError}} \\
\hline
\Xi \mathit{OrderSystem} \\
\mathit{id}? : \mathit{ORDERID} \\
\mathit{report}! : \mathit{REPORT} \\
\hline
\neg\ (\mathit{orders}(\mathit{id}?) \sqsubseteq \mathit{stock}) \\
\mathit{report}! = \mathit{not\_enough\_stock} \\
\hline
\end{array}
$$

We define operation *InvoiceOrderOp* to deal with all situations. When multiple errors happen at the same time, the definition is unspecific about which error report to return. We will discuss nondeterminism in Chapter 8.

$$
\begin{aligned}
\mathit{InvoiceOrderOp} \mathrel{\widehat{=}} {} & \mathit{InvoiceOrder} \lor \mathit{IdNotFoundError} \lor \\
& \qquad \mathit{OrderNotPendingError} \lor \mathit{NotEnoughStockError}
\end{aligned}
$$

3.5 Cancel Order

A *pending* order may be cancelled. The cancelled order id will be made available for new orders in the future. Status *pending* is a pre-condition to cancel order 2_a in the scenario on Table 3.12. The first predicate in schema *CancelOrder* captures this pre-condition. The third and fourth predicates capture the disappearance of 2_a from $orders'$ and $orderStatus'$ where the domain anti-restriction symbol $\lhd$ is used to remove maplets of $id?$. The second last predicate captures the addition of 2_a to $freeIds'$.

$id? = 2_a$

stock	*orders*	*orderStatus*	*freeIds*
$\{nut \mapsto 5,$ $bolt \mapsto 6\}$	$\{1 \mapsto \{nut \mapsto 2, bolt \mapsto 2\},$ $2_a \mapsto \{bolt \mapsto 3\}\}$	$\{1 \mapsto invoiced,$ $2_a \mapsto pending\}$	$\{3, 4\}$
...	$\{1 \mapsto \{nut \mapsto 2, bolt \mapsto 2\}\}$	$\{1 \mapsto invoiced\}$	$\{2_a\ , 3, 4\}$

$report! = OK$

Tab. 3.12: **Z-scenario** *CancelOrder*

CancelOrder

$\Delta OrderSystem$
$id? : ORDERID$
$report! : REPORT$

$orderStatus(id?) = pending$
$stock' = stock$
$orders' = \{id?\} \lhd orders$
$orderStatus' = \{id?\} \lhd orderStatus$
$freeIds' = \{id?\} \cup freeIds$
$report! = OK$

It is futile to cancel an order that does not exist or has been *invoiced*. We reuse two error detecting schemas to handle those situations.

$$CancelOrderOp \mathrel{\widehat{=}} CancelOrder \lor IdNotFoundError \lor OrderNotPendingError$$

3.6 Enter Stock

Table 3.13 describes Z-scenario *EnterStock* for the replenishment of depleted stock.

$newStock? = \{nut \mapsto 80, bolt \mapsto 70\}_a$

$stock$	$orders$	$orderStatus$	$freeIds$
$\{nut \mapsto 5,$ $bolt \mapsto 9\}_b$	$\{1 \mapsto \{nut \mapsto 2, bolt \mapsto 2\},$ $2 \mapsto \{bolt \mapsto 3\}\}$	$\{1 \mapsto invoiced,$ $2 \mapsto pending\}$	$\{3, 4\}$
$\{nut \mapsto 85,$ $bolt \mapsto 79\}_c$	$\ldots$	$\ldots$	$\ldots$

$report! = OK \wedge$

$$\{nut \mapsto 85, bolt \mapsto 79\}_c = \{nut \mapsto 5, bolt \mapsto 9\}_b \uplus \{nut \mapsto 80, bolt \mapsto 70\}_a$$

Tab. 3.13: **Z-scenario** *EnterStock*

Symbol $\uplus$ in the postcondition stands for pairwise bag additions.

EnterStock

$\Delta OrderSystem$

$newStock? : ORDER$

$report! : REPORT$

$stock' = stock \uplus newStock?$

$orders' = orders$

$orderStatus' = orderStatus$

$freeIds' = freeIds$

$report! = OK$

For simplification, we assume that the warehouse is so large that new stock cannot lead to error.

3.7 Underspecification

Our approach to generalise scenarios to schemas is not foolproof. We may overlook a condition resulting in underspecification. The scenarios generated from an incorrect schema allow us to detect underspecification. Suppose we had omitted the following precondition in schema *InvoiceOrder* on Page 59 that checks for sufficient stock.

$$orders(id?) \sqsubseteq stock$$

When we try to invoice an order for six nuts on a stock of 5 nuts with the underspecified schema, we will get the scenario in Table 3.14. Take note of the three underlined quantities of *nuts*.

$id? = 2$

$stock$	$orders$	$orderStatus$	$freeIds$
$\{nut \mapsto \underline{5},$ $bolt \mapsto 9\}$	$\{1 \mapsto \{nut \mapsto 2\},$ $2 \mapsto \{nut \mapsto \underline{6},$ $bolt \mapsto 6\}\}$	$\{1 \mapsto invoiced,$ $2 \mapsto pending\}$	$\{3, 4\}$
$\{nut \mapsto \underline{0},$ $bolt \mapsto 3\}$	...	$\{1 \mapsto invoiced,$ $2 \mapsto invoiced\}$	...

$report! = OK$

Tab. 3.14: **Z-scenario** *UnderSpecifiedInvoiceOrder*

According to the definition of bag difference operator $\uminus$ in *Z Reference Manual* [133, page 126], the result is zero when the subtrahend is greater than the minuend. But $5 - 6 = 0$ is wrong. Even if we are ignorant about the effect of operator $\uminus$, the result of $5 - 6 = -1$ will still raise an eyebrow because it is impossible to have negative stock. We learn from this test that the schema is wrong.

3.8 Overspecification

Overspecification happens when unnecessary conditions are included in a schema. Recall scenario *CreateOrder* on page 55. The quantities of the nuts and bolts in the input parameter were both 4 by coincidence.

$order? = \{nut \mapsto 4_a \ , bolt \mapsto 4_a \ \}$

$stock$	$orders$	$orderStatus$	$freeIds$
$\{nut \mapsto 5,$ $bolt \mapsto 6\}$	$\{1 \mapsto \{nut \mapsto 2, bolt \mapsto 2\},$ $2 \mapsto \{bolt \mapsto 3\}\}$	$\{1 \mapsto invoiced,$ $2 \mapsto invoiced\}$	$\{3, 4\}$
$\ldots$	$\{1 \mapsto \{nut \mapsto 2, bolt \mapsto 2\},$ $2 \mapsto \{bolt \mapsto 3\},$ $3 \mapsto \{nut \mapsto 4_a \ , bolt \mapsto 4_a \ \}\}$	$\{1 \mapsto invoiced,$ $2 \mapsto invoiced,$ $3 \mapsto pending\}$	$\{4\}$

$newId! = 3, report! = OK$

Tab. 3.15: **Z-scenario** *OverSpecifiedCreateOrder*

If the specifier had mistaken it as a general rule, the equality may be codified as the first predicate in the schema on this page. The predicate evaluates to false on a legitimate order for $\{nut \mapsto 7, bolt \mapsto 9\}$. The overspecification is caught.

OverSpecifiedCreateOrder

$\Delta OrderSystem$
$order? : ORDER$
$newId! : ORDERID$
$report! : REPORT$

$\forall p, q : PRODUCT \mid p \in \operatorname{dom} order? \land q \in \operatorname{dom} order? \bullet$
$\qquad order?(p) = order?(q)$
$newId! \in freeIds$
$stock' = stock$
$orders' = orders \cup \{newId! \mapsto order?\}$
$orderStatus' = orderStatus \cup \{newId! \mapsto pending\}$
$freeIds' = freeIds \setminus \{newId!\}$
$report! = OK$

3.9 Totality and Coverage

Given our scenario-driven approach, an obvious question to ask is whether we have covered enough scenarios. The specifier needs to make sure that the operations are total. An operation is **total** if its precondition is always true. In other words, a total operation handles all situations. We have defined the combination of all four operations to be total. Recall the definition of *InvoiceOrderOp*.

$$\mathit{InvoiceOrderOp} \mathrel{\widehat{=}} \mathit{InvoiceOrder} \lor \mathit{IdNotFoundError} \lor \\ \mathit{OrderNotPendingError} \lor \mathit{NotEnoughStockError}$$

Let A be the precondition of *InvoiceOrderOp*
B ˜ *InvoiceOrder*
C ˜ *IdNotFoundError*
D ˜ *OrderNotPendingError*
E ˜ *NotEnoughStockError*

C = $id? \notin \operatorname{dom} \mathit{orderStatus}$
D = $\mathit{orderStatus}(id?) \neq \mathit{pending}$
E = $\lnot\ (\mathit{orders}(id?) \sqsubseteq \mathit{stock})$
B = $id? \in \operatorname{dom} \mathit{orderStatus} \land \mathit{orderStatus}(id?) = \mathit{pending} \land \mathit{orders}(id?) \sqsubseteq \mathit{stock}$

With the help of DeMorgan's Laws, we prove the totality of *InvoiceOrderOp* as follows.

$$\mathrm{B} = \lnot\ C \land \lnot\ D \land \lnot\ E$$
$$\mathrm{B} = \lnot\ (C \lor D) \land \lnot\ E$$
$$\mathrm{B} = \lnot\ (C \lor D \lor E)$$

$$\mathrm{A} = B \lor C \lor D \lor E$$
$$\mathrm{A} = \lnot\ (C \lor D \lor E) \lor (C \lor D \lor E)$$
$$\mathrm{A} = \mathit{true}$$

A total operation handles all situations but not necessarily in a way desired by the customer. Totality is necessary but insufficient for an operation to fully meet customer requirements. The use of scenarios cannot completely eliminate the possibility of an incomplete formal specification. But this is not a problem caused by using scenarios. A formal specification may be incomplete regardless of the approach used to build it. However the use of scenarios to encourage customer participation can only improve our odds of having a complete formal specification.

We do not want our schemas to work on only a few scenarios. We need to have correct and complete conditions captured in the predicates. The checking of underspecification, overspecification and totality helps.

3.10 Understandability to Customers

Biologists and educationists believe that understanding begins with concrete examples [150, pages 102–103]. Thanks to their training, formalism experts are comfortable with abstraction used in specifications. Our scenarios are concrete examples which customers would find easier to understand than abstract specifications.

Z-scenarios are written in few symbols denoting simple concepts. The Z-scenarios in our example only use a small number of symbols: $? \; ! = \{ \; \} \mapsto$. The formal specification uses all the above symbols plus many more, such as $\Delta \; \Xi \; == \; | \; \widehat{=} \; \mathbf{bag} \; ⇸ \; \mathbf{dom} \; \cup \; \cap \; \in \; \setminus \; \emptyset \; \sqsubseteq \; \uplus \; \oplus \; \notin \; \neg \; \wedge \; \vee \; \forall$. Scenarios only list concrete values while specifications use expressions constructed from variables. We expect the phenomenon of scenarios using fewer and simpler symbols to extend to other problem domains. Scenarios are simpler than schemas.

If the customer still finds Z-scenarios too difficult, he or she can get help from the E-scenarios. Commentary in specifications is not as helpful as E-scenarios. The use of variables and additional concepts makes it harder to write accurate English descriptions for formal specifications. Once written, the commentary will be more difficult to understand than E-scenarios again due to the additional concepts used.

Since each scenario only deals with a user task in a situation, many scenarios may be needed. Appropriate tool support can help us deal with large numbers of scenarios much like the way test cases are managed. If we use scenarios for the double duty of guiding and verifying software development, the costs of using them will be lowered.

Scenarios in this chapter only have a single step. In later chapters, we will work with scenarios of multiple steps. When a scenario gets too long, we may break it up into two or more shorter scenarios. Even for a long scenario, the customer can understand it one step at a time. On balance scenarios are more understandable to the customers than formal specifications.

3.11 Chapter Conclusion

Our approach can be described as a process of creating a few artefacts. The sentence templates and E-scenarios are the responsibilities of customers. The English to Z translation rules, type definitions, system state schemas, Z-scenarios and schemas are the responsibilities of developers. The artefacts embody many decisions. Our examples in the thesis represent the decisions we make in the roles of customers and developers. Different decisions will lead to different solutions. The issue of efficiency aside, our schemas should always meet the requirements captured in the scenarios. In Chapter 5 on specification maintenance, we will see how a previously acceptable decision on the system state turns out to be a bad one as the requirements evolve. To retract the decision, we have to revise many schemas.

Customer involvement is a critical factor of software projects [61]. The use of concrete scenarios allows customers to participate more directly in the creation of formal specifications.

A set of scenarios is a partial rather than complete specification. Through the generalisation of data values and their relationships into variables and predicates, we create a complete specification with Z schemas. Additional scenarios can be used to validate a Z specification to gain confidence in its correctness and completeness.

4. SPECIFICATION DEVELOPMENT

Scenarios partition functional requirements. Customers can rank scenarios by their importance or urgency. This ranking suggests an order in which the scenarios should be tackled. Developers work on a small number of scenario steps at one time without the risk of being overwhelmed. For example, in this chapter, we will work on the most common scenarios before the rare ones.

Our example is a telephone system chosen partly for the interactions of multiple phone users. Contrary to the one-step scenarios in the previous chapter, interactions require multiple steps in a scenario.

Section 4.1 describes the features expected by the customer. Sections 4.2 and 4.3 list the sentence templates for the state space and input/output parameters. Their translations to Z notation are shown. Section 4.4 shows a simple E-scenario and its equivalent Z-scenario. Sections 4.5 to 4.7 derive the basic operations of dialling, answering and hanging. Section 4.8 considers a scenario of competing calls and no answer. Section 4.9 proves the totality and determinism of the dialling operation. Section 10 concludes the iterative development approach we just demonstrated.

4.1 Telephone System

We specify a primitive phone system to resemble the way mobile phones work. The interoperability with landline phones is excluded lest the details distract readers from the main message. There are three basic operations: dialling, answering and hanging. The *dialling* operation is ignored if the caller is connected or being rung. Otherwise it results in a ring or busy tone depending on the state of the callee. The *answering* operation can be successfully performed by someone whose phone is ringing. It results in a new connection. When performed in other states, the answering operation is ignored. The *hanging* operation can be successfully performed by someone who is engaged in a connection. Performed in other states, the answering operation is ignored.

Readers may find the examples in this thesis verbose. The verbiage reflects our attempt to accommodate newcomers to the approach.

4.2 State Space

To keep the presentation compact, we only use single-digit phone numbers from 1 to 9.

$$PHONE == \{phone : 1 \mathinner{..} 9\}$$

In Table 4.1, we keep two kinds of information in the system state: connection and ring.

States in English	**Z Expressions**
No. $\underline{1}$ is connected to no. $\underline{2}$.	$connection = \{1 \mapsto 2, 3 \mapsto 4\}$
No. $\underline{3}$ is connected to no. $\underline{4}$.	
No. $\underline{1}$ is ringing no. $\underline{2}$.	$ringing = \{1 \mapsto 2\}$

Tab. 4.1: Telephone System States – from English to Z

Schema *PhoneSystem* stores the system state in variables *connection* and *ringing*. We define the variables as partial functions with the $\mathrel{+\!\!\!\!\rightarrow}$ symbol. If *ringing* were defined as a total function by mistake, every phone would be ringing all the time.

PhoneSystem

$$connection, ringing : PHONE \mathrel{+\!\!\!\!\rightarrow} PHONE$$

It is a common practice for Z experts to write invariants of the system state at this point. For example, a phone cannot be simultaneously rung by multiple callers. We deem the explicitly stated invariants optional because the system behaviour is sufficiently constrained by concrete scenarios and their generalised schemas. On the other hand, stating the invariants can enhance communication. In practice, we can only write state invariants after having considered enough scenarios.

Sentence templates are decided by customers. Variable names, such as *connection* and *ringing*, are decided by developers. Other names could be used. For readability, it is best to choose the names meaningful in the application domain.

4.3 Input/Output

After describing the system state, customers work with specifiers on the user interface. In this example, interactions are initiated by users. Table 4.2 shows three kinds of user input.

Input in English	**Z Expressions**
User at no. 3 dials to no. 4	$\text{caller?} = 3 \land \text{callee?} = 4$
User at no. 4 answers the phone	$\text{answer?} = 4$
User at no. 3 hangs up the phone	$\text{hang?} = 3$

Tab. 4.2: Telephone System Input – from English to Z

Shown in Table 4.3, the system returns three kinds of responses on the dialling operation, a ring, busy tone or no tone at all. For the operations of answering or hanging, a user gets an *okay* or *ignored* result.

Output in English	**Z Expressions**
User gets a ring	$\text{tone!} = \text{ring}$
User gets a busy tone	$\text{tone!} = \text{busy}$
User gets no tone	$\text{tone!} = \text{silent}$
Request is OK	$\text{rqt!} = \text{OK}$
Request is ignored	$\text{rqt!} = \text{ignored}$

Tab. 4.3: Telephone System Outputs – from English to Z

We define data types *TONE* and *RESULT* for the possible values on output parameters *tone*! and *rqt*!.

$$TONE \quad ::= ring \mid busy \mid silent$$

$$RESULT ::= OK \mid ignored$$

Interactions are user observable events. For example, dialling a number involves the pressing of numeric keys and the *send* button on a mobile phone. Disconnecting a call involves the pressing of the *end* button. Customers decide the level of details they want to capture. In our case, we have chosen to ignore the pressing of individual numeric keys.

4.4 Scenarios of Simple Calls

Steps are the building blocks of concrete scenarios. A step consists of input and output parameters. It transforms the system from one state to the next. For an easy introduction to concrete scenarios, we only presented single-step scenarios in the previous chapter. Multi-step scenarios are required to show the interactions of two or more phone users. The adapted scenario table should be read from left to right and then top to bottom. The first column holds the step and state numbers. In the following scenario, the system is initially in state 0 where users at no. 3 and no. 4 are already connected. After the caller at no. 1 dials no. 2 to get a ring tone, the system is in the state where no. 1 is ringing no. 2. After the callee has picked up the phone, the system now has two active connections. After the phone user at no. 1 has hung up, the system is back to a state with only one connection.

Step	**Input/Output**	**System State**
0		No. 3 is connected to no. 4.
1	User at no. 1 dials to no. 2. User gets a ring tone.	No. 3 is connected to no. 4. No. 1 is ringing no. 2.
2	User at no. 2 answers the phone. Request is OK.	No. 3 is connected to no. 4. No. 1 is connected to no. 2.
3	User at no. 1 hangs up the phone. Request is OK.	No. 3 is connected to no. 4.

Tab. 4.4: **E-scenario** *MakeSimpleCall*

Specifiers or automated tools translate E-scenarios in English to their equivalent Z-scenarios in Z notation. Whenever required by the customers for better comprehension, the Z-scenario in say Table 4.5 on the following page can be reverted back to the original E-scenario in Table 4.4. The readers of the thesis are expected to be familiar with formal notations. We will work mainly with the more succinct Z-scenarios.

Step	**Input**	**Output**	***connection***	***ringing***
0			$\{3 \mapsto 4\}$	$\{\ \}$
1	$caller? = 1 \land callee? = 2$	$tone! = ring$	$\{3 \mapsto 4\}$	$\{1 \mapsto 2\}$
2	$answer? = 2$	$rqt! = OK$	$\{1 \mapsto 2, 3 \mapsto 4\}$	$\{\ \}$
3	$hang? = 1$	$rqt! = OK$	$\{3 \mapsto 4\}$	$\{\ \}$

Tab. 4.5: **Z-scenario** *MakeSimpleCall*

The steps of Z-scenario *MakeSimpleCall* represent three operations each with its own parameter names and types. We use the dot notation to refer to individual steps of a scenario. For example, we write *MakeSimpleCall*.1 to refer to the first step in the scenario.

Specifiers discover patterns of data relationships in scenario steps and generalise them to predicates. It helps to consider a pair of complementary steps together as we will do in the following sections. Steps are said to be complementary if they represent the same operation in different situations. The scenario steps used for generalisation may influence the patterns initially discovered. By considering more scenario steps for the operation, as was done in Sections 3.7 and 3.8, we should eventually arrive at more or less the same operation schema.

In the previous chapter, we worked with many numbers for stock or order quantities. There may be equalities in the scenarios. We used subscripts to distinguish required equalities from coincidental equalities. In this chapter, all phone numbers are unique. All equalities are required. We need no subscripts for differentiation.

4.5 Dialling Operation

Schema *DialRing* generalizes a successful dialling attempt documented in step *MakeSimpleCall*.1. $\Delta PhoneSystem$ declares that its variables may be changed by the operation. Variable names without and with a prime stand for values before and after the operation respectively. Caller 1 and callee 2 are not engaged in any connection or ringing activity. The observations are captured by the first two predicates. The fourth predicate uses override symbol $\oplus$ to state that the new value of *ringing* will be the same as before except for $caller? \mapsto callee?$.

DialRing

$\Delta PhoneSystem$
$caller?, callee? : PHONE$
$tone! : TONE$

$caller? \notin \mathrm{dom}\ connection \cup \mathrm{ran}\ connection \cup \mathrm{dom}\ ringing \cup \mathrm{ran}\ ringing$
$callee? \notin \mathrm{dom}\ connection \cup \mathrm{ran}\ connection \cup \mathrm{dom}\ ringing \cup \mathrm{ran}\ ringing$
$connection' = connection$
$ringing' = ringing \oplus \{caller? \mapsto callee?\}$
$tone! = ring$

DialBusy.1 is a step that gets the busy tone on a dialling operation. The value 2 of $callee?$ appears in the *connection* of the pre-state of step *DialBusy*.1. After consulting with the customer, the developer wrote the second predicate in schema *DialBusy* on the following page to capture this fact. This predicate is the negation of the corresponding predicate in schema *DialRing*. Variables *connection* and *ringing* in *PhoneSystem* are unchanged by this operation as indicated by a preceding Xi symbol Ξ.

Step	**Input**	**Output**	***connection***	***ringing***
0			$\{2 \mapsto 3\}$	$\{\ \}$
1	$caller? = 1 \wedge callee? = 2$	$tone! = busy$	$\{2 \mapsto 3\}$	$\{\ \}$

Tab. 4.6: **Z-scenario** *DialBusy*

$$
\begin{array}{|l}
\multicolumn{1}{l}{\mathit{DialBusy}} \\
\hline
\Xi \mathit{PhoneSystem} \\
\mathit{caller}?, \mathit{callee}? : \mathit{PHONE} \\
\mathit{tone}! : \mathit{TONE} \\
\hline
\mathit{caller}? \notin \operatorname{dom} \mathit{connection} \cup \operatorname{ran} \mathit{connection} \cup \operatorname{dom} \mathit{ringing} \cup \operatorname{ran} \mathit{ringing} \\
\mathit{callee}? \in \operatorname{dom} \mathit{connection} \cup \operatorname{ran} \mathit{connection} \cup \operatorname{dom} \mathit{ringing} \cup \operatorname{ran} \mathit{ringing} \\
\mathit{tone}! = \mathit{busy} \\
\hline
\end{array}
$$

Step *DialIgnored*.1 is an ignored dialling operation. The precondition of *callee*? appearing in *connection* is generalised to the first predicate in schema *DialIgnored* which also checks for callee's appearance in *ringing*.

Step	**Input**	**Output**	***connection***	***ringing***
0			$\{1 \mapsto 3\}$	$\{ \}$
1	$\mathit{caller}? = 1 \land \mathit{callee}? = 2$	$\mathit{tone}! = \mathit{silent}$	$\{1 \mapsto 3\}$	$\{ \}$

Tab. 4.7: **Z-scenario** *DialIgnored*

$$
\begin{array}{|l}
\multicolumn{1}{l}{\mathit{DialIgnored}} \\
\hline
\Xi \mathit{PhoneSystem} \\
\mathit{caller}?, \mathit{callee}? : \mathit{PHONE} \\
\mathit{tone}! : \mathit{TONE} \\
\hline
\mathit{caller}? \in \operatorname{dom} \mathit{connection} \cup \operatorname{ran} \mathit{connection} \cup \operatorname{dom} \mathit{ringing} \cup \operatorname{ran} \mathit{ringing} \\
\mathit{tone}! = \mathit{silent} \\
\hline
\end{array}
$$

$$\mathit{DiallingOp} \mathrel{\widehat{=}} \mathit{DialRing} \lor \mathit{DialBusy} \lor \mathit{DialIgnored}$$

4.6 Answering Operation

AnswerIgnored.1 is an ignored answering request which complements the successful attempt in step *MakeSimpleCall*.2 in Table 4.5 on page 73.

Step	**Input**	**Output**	***connection***	***ringing***
0			$\{3 \mapsto 4\}$	$\{\ \}$
1	$answer? = 2$	$rqt! = ignored$	$\{3 \mapsto 4\}$	$\{\ \}$

Tab. 4.8: **Z-scenario** *AnswerIgnored*

MakeSimpleCall.2 is an answering operation resulting in a connection of two phones. We generalise the step to schema *AnswerRing* which has a local variable *caller*. The first predicate determines the caller causing the ring. The second predicate updates *connection* with $caller \mapsto answer?$. The third predicate uses the range subtraction symbol $\rhd$ to update function *ringing*. It removes the ring for phone no. *answer?*.

$$\begin{array}{|l}
\hline
\textit{AnswerRing} \\
\Delta \mathit{PhoneSystem} \\
answer? : \mathit{PHONE} \\
rqt! : \mathit{RESULT} \\
caller : \mathit{PHONE} \\
\hline
caller \mapsto answer? \in ringing \\
connection' = connection \oplus \{caller \mapsto answer?\} \\
ringing' = ringing \rhd \{answer?\} \\
rqt! = OK \\
\hline
\end{array}$$

Schema *AnswerIgnored* generalizes step *AnswerIgnored*.1.

AnswerIgnored

$\Xi PhoneSystem$

$answer? : PHONE$

$rqt! : RESULT$

$answer? \notin \operatorname{ran} ringing$

$rqt! = ignored$

$AnsweringOp \mathrel{\widehat=} AnswerRing \lor AnswerIgnored$

4.7 Hanging Operation

HangIgnored.1 is an ignored hang up request which corresponds to the event of a user pressing the end button on a mobile phone when it is not connected. The step is complementary to *MakeSimpleCall*.3 in Table 4.5 on page 73.

Step	**Input**	**Output**	***connection***	***ringing***
0			$\{3 \mapsto 4\}$	$\{\,\}$
1	$hang? = 2$	$rqt! = ignored$	$\{3 \mapsto 4\}$	$\{\,\}$

Tab. 4.9: **Z-scenario** *HangIgnored*

Schema *HangConnect* generalizes step *MakeSimpleCall*.3 of terminating a connection. Function *connection* is updated by removing the ordered pair that contains the hanging phone number. Since we do not know whether the hanging phone number is the caller or callee, we may as well apply both domain and range subtraction, denoted by ⩤ and ⩥ symbols respectively. If say the input phone number is in the range but not in the domain, the range subtraction will remove the ordered pair while the domain subtraction will have no effect.

$$
\begin{array}{|l}
\multicolumn{1}{l}{\textit{HangConnect}} \\
\hline
\Delta \textit{PhoneSystem} \\
\textit{hang}? : \textit{PHONE} \\
\textit{rqt}! : \textit{RESULT} \\
\hline
\textit{hang}? \in \operatorname{dom} \textit{connection} \cup \operatorname{ran} \textit{connection} \\
\textit{connection}' = \{\textit{hang}?\} \ndres \textit{connection} \nrres \{\textit{hang}?\} \\
\textit{ringing}' = \textit{ringing} \\
\textit{rqt}! = \textit{OK} \\
\hline
\end{array}
$$

Schema *HangIgnored* generalizes step *HangIgnored*.1.

$$
\begin{array}{|l}
\multicolumn{1}{l}{\textit{HangIgnored}} \\
\hline
\Xi \textit{PhoneSystem} \\
\textit{hang}? : \textit{PHONE} \\
\textit{rqt}! : \textit{RESULT} \\
\hline
\textit{hang}? \notin \operatorname{dom} \textit{connection} \cup \operatorname{ran} \textit{connection} \\
\textit{rqt}! = \textit{ignored} \\
\hline
\end{array}
$$

$$\textit{HangingOp} \mathrel{\widehat{=}} \textit{HangConnect} \lor \textit{HangIgnored}$$

4.8 Competing Calls

In the previous sections, we include scenarios *MakeSimpleCall*, *DialBusy*, *AnswerIgnored* and *HangIgnored* in our first iteration because they are the most common scenarios. This section represents our second iteration where we consider the less common scenario of two parties calling the same number at about the same time. The first caller gets a ring tone; the second caller gets a busy tone. Since the callee does not pick up the phone, the first caller hangs up before a connection is ever made.

Before we modify the Z specification for a new step, we want to see if an existing schema already handles to the step. Steps *CompetingCalls*.1 and *MakeSimpleCall*.1 are identical. Schema *DialRing* that works on one must also work on the other. No modification to the schema is required.

Z-scenario *CompetingCalls*

Step	**Input**	**Output**	***connection***	***ringing***
0			$\{3 \mapsto 4\}$	$\{\ \}$
1	$caller? = 1 \land callee? = 2$	$tone! = ring$	$\{3 \mapsto 4\}$	$\{1 \mapsto 2\}$
2	$caller? = 5 \land callee? = 2$	$tone! = busy$	$\{3 \mapsto 4\}$	$\{1 \mapsto 2\}$
3	$hang? = 1$	$rqt! = OK$	$\{3 \mapsto 4\}$	$\{\ \}$

Tab. 4.10: **Z-scenario** *CompetingCalls*

4.8.1 Callee Already Ringing

Next we try to see if step *CompetingCalls*.2 can be handled by the current specification. The I/O parameters of the step tell us that it is a dialling operation. We shall apply the step's data substitutions to the three disjuncts of *DiallingOp*. The second disjunct schema *DialBusy* from page 75 has three predicates, the first two are precondition and the last one is postcondition. All three predicates evaluate to true as shown below. It means that the schema handles this step perfectly.

$$\begin{array}{l}
caller? \notin \mathrm{dom}\, connection \cup \mathrm{ran}\, connection \cup \mathrm{dom}\, ringing \cup \mathrm{ran}\, ringing \\
\Leftrightarrow 5 \notin \mathrm{dom}\{3 \mapsto 4\} \cup \mathrm{ran}\{3 \mapsto 4\} \cup \mathrm{dom}\{1 \mapsto 2\} \cup \mathrm{ran}\{1 \mapsto 2\} \\
\Leftrightarrow 5 \notin \{3\} \cup \{4\} \cup \{1\} \cup \{2\} \\
\Leftrightarrow 5 \notin \{1, 2, 3, 4\} \\
\Leftrightarrow true
\end{array}$$

$$\begin{array}{l}
callee? \in \mathrm{dom}\, connection \cup \mathrm{ran}\, connection \cup \mathrm{dom}\, ringing \cup \mathrm{ran}\, ringing \\
\Leftrightarrow 2 \in \{1, 2, 3, 4\} \\
\Leftrightarrow true
\end{array}$$

$$\begin{array}{l}
tone! = busy \\
\Leftrightarrow busy = busy \\
\Leftrightarrow true
\end{array}$$

4.8.2 No Answer

We check if the current specification handles step *CompetingCalls*.3 on the previous page. The input parameter *hang*? tells us that it is a hanging operation. *HangingOp* is a disjunction of *HangConnect* and *HangIgnored*. Though not shown here, after applying substitutions from the step, each of the schemas has a predicate concerning the value of *ringing* evaluate to false. We need a new schema *HangRing* to specify the correct value for *ringing*. According to the step, *hang*? is in the domain of *ringing*. In other words, the input phone number must be ringing someone. We express this precondition as the first predicate in the new schema. The second predicate removes from function *ringing* the ordered pair with the first phone number equal to *hang*?. It stops the ringing caused by *hang*?. The schema handles the case where the caller hangs up before the callee picks up the phone.

$$\begin{array}{|l}
\multicolumn{1}{l}{\textit{HangRing}} \\
\hline
\Delta PhoneSystem \\
hang? : PHONE \\
rqt! : RESULT \\
\hline
hang? \in \operatorname{dom} ringing \\
ringing' = \{hang?\} \mathbin{\lhd\!\!\!-} ringing \\
connection' = connection \\
rqt! = OK \\
\hline
\end{array}$$

The precondition of existing schema *HangIgnored* is also true for step *CompetingCalls*.3. This is a problem because its postcondition does not match the step. We strengthen the precondition so that it evaluates to false for the step.

$$\begin{array}{|l}
\multicolumn{1}{l}{\textit{HangIgnored}} \\
\Xi \textit{PhoneSystem} \\
\textit{hang}? : \textit{PHONE} \\
\textit{rqt}! : \textit{RESULT} \\
\hline
\textit{hang}? \notin \mathrm{dom}\ \textit{connection} \cup \mathrm{ran}\ \textit{connection} \cup \mathrm{dom}\ \textit{ringing} \\
\textit{rqt}! = \textit{ignored} \\
\hline
\end{array}$$

HangingOp is redefined to include new schema *HangRing*.

$$\textit{HangingOp} \mathrel{\widehat{=}} \textit{HangConnect} \lor \textit{HangRing} \lor \textit{HangIgnored}$$

4.9 Totality and Determinism

An operation is *total* if its precondition is true. A total operation handles all situations though not necessarily in a way desired by the customers. Totality is a necessary but not a sufficient condition for an operation to meet all customer requirements. In layman's terms, we make sure that our operation specification is total but we are not content yet.

An operation is *deterministic* if no two of its disjunct constituents overlap in their preconditions. When there is an overlap, either schema can engage in a step. Unless both schemas achieve the same outcome, users will be confused by the differing results from the same input and pre-state. We devote Chapter 8 to nondeterminism.

DiallingOp, *AnsweringOp* and *HangingOp* are total and deterministic. The totality and determinism of *DiallingOp* are justified below.

Let the preconditions of *DialRing*, *DialBusy*, *DialIgnored* and *DiallingOp* be **P**, **Q**, **R** and **S** respectively.

Let $\mathbf{a} = \textit{caller}? \in \mathrm{dom}\ \textit{connection} \cup \mathrm{ran}\ \textit{connection} \cup \mathrm{dom}\ \textit{ringing} \cup \mathrm{ran}\ \textit{ringing}$

$\mathbf{b} = \textit{callee}? \in \mathrm{dom}\ \textit{connection} \cup \mathrm{ran}\ \textit{connection} \cup \mathrm{dom}\ \textit{ringing} \cup \mathrm{ran}\ \textit{ringing}$

$\mathbf{P} = \neg\, \mathbf{a} \land \neg\, \mathbf{b}$

$\mathbf{Q} = \neg\, \mathbf{a} \land \mathbf{b}$

$\mathbf{R} = \mathbf{a}$

$$
\begin{aligned}
\mathbf{S} &= \mathbf{P} \vee \mathbf{Q} \vee \mathbf{R} \\
&= (\neg\, \mathbf{a} \wedge \neg\, \mathbf{b}) \vee (\neg\, \mathbf{a} \wedge \mathbf{b}) \vee \mathbf{a} \\
&= (\neg\, \mathbf{a} \wedge (\neg\, \mathbf{b} \vee \mathbf{b})) \vee \mathbf{a} \\
&= (\neg\, \mathbf{a} \wedge \mathit{true}) \vee \mathbf{a} \\
&= \neg\, \mathbf{a} \vee \mathbf{a} \\
&= \mathit{true}
\end{aligned}
$$

$\therefore$ *DiallingOp* is a total operation.

$$
\begin{aligned}
\mathbf{P} \wedge \mathbf{Q} &= (\neg\, \mathbf{a} \wedge \neg\, \mathbf{b}) \wedge (\neg\, \mathbf{a} \wedge \mathbf{b}) \\
&= \neg\, \mathbf{a} \wedge (\neg\, \mathbf{b} \wedge \mathbf{b}) \\
&= \neg\, \mathbf{a} \wedge \mathit{false} \\
&= \mathit{false}
\end{aligned}
$$

$$
\begin{aligned}
\mathbf{P} \wedge \mathbf{R} &= (\neg\, \mathbf{a} \wedge \neg\, \mathbf{b}) \wedge \mathbf{a} \\
&= \neg\, \mathbf{a} \wedge \mathbf{a} \wedge \neg\, \mathbf{b} \\
&= \mathit{false} \wedge \neg\, \mathbf{b} \\
&= \mathit{false}
\end{aligned}
$$

$$
\begin{aligned}
\mathbf{Q} \wedge \mathbf{R} &= (\neg\, \mathbf{a} \wedge \mathbf{b}) \wedge \mathbf{a} \\
&= \neg\, \mathbf{a} \wedge \mathbf{a} \wedge \mathbf{b} \\
&= \mathit{false} \wedge \mathbf{b} \\
&= \mathit{false}
\end{aligned}
$$

$\therefore$ *DiallingOp* is deterministic.

4.10 Chapter Conclusion

Customers represent domain concepts at their desired level of details. If our customer was a telephone set manufacturer, he would be interested in the pressing of individual keys. Customers and specificiers work together to write concrete scenarios using these domain concepts in English constrained to a small set of sentence templates. Specifiers translate scenarios in English to equivalent scenarios in Z notation.

Writing the formal specification for a large system can be overwhelming. Developers need to decide where to start. Concrete scenarios should be ranked by customers for importance. Prioritised scenarios provide good guidance. At any time, developers work on the more important scenarios that remain.

An operation is defined as a disjunction of schemas. A disjunct schema handles a family of similar situations. For a given step, we would normally only want one disjunct's precondition to be true so that other disjuncts do not interfere. The precondition of an existing schema can be weakened to handle a new step if the postcondition of the schema matches the step. If no disjunct schema has a matching postcondition, we create a new schema. During specification maintenance, we identify different groups of predicates that constitute the precondition and the postcondition. The Z notation does not declare whether a predicate belongs to the precondition or postcondition but it is not difficult to differentiate them. The precondition is captured by the predicates that refer to post-state variables or output parameters. All other predicates constitute the postcondition.

Scenarios are an organisation tool. We can maintain traceability between a step and its handling schema. Any time a schema is modified, the developer should rerun an animator to verify the updated formal specification against the concrete scenarios. Working with concrete scenarios would be similar to working with test cases, good tool support is crucial to manage them in numbers.

This chapter comprises two iterations. The first iteration deals with the basic functionality. The second iteration deals with competing calls. We can prove that the schemas from the first iteration are total and deterministic. Totality gives us the false impression that a formal specification is correct and complete. Concrete scenarios help us to discover missing functionality in a formal specification.

5. SPECIFICATION MAINTENANCE

During the initial development of a formal specification, we would consider two or more complementary steps together in the writing of the disjunct schemas for an operation. For example, based on the steps that return busy tone and ring tone, we write schemas *DialRing* and *DialBusy* to define *DiallingOp*.

During the maintenance of a formal specification, we often consider new scenarios one step at a time resulting in the modifications and/or additions of schemas. The second iteration in the last chapter was carried out just like a maintenance effort. In this chapter, we work on a sizeable change in customer requirements that warrants an update to the state representation. In order to support conference calls, the representation of phone connections is changed from ordered pairs to sets. We revise the existing schemas to work with the new state representation before working on the new scenarios.

The theme of the chapter is to describe the use of scenarios to facilitate specification maintenance. Section 5.1 gives an account of the scanty existing work. Section 5.2 describes a change to the state space for conferencing. Section 5.3 revises the existing schemas for the new state space. Section 5.4 creates the schemas to start and end a conference. We also strengthen the preconditions of existing schemas so that they do not interfere with the new schemas. Section 5.5 discusses the necessary reverification after scenarios and schemas are updated. Section 5.6 concludes the chapter with a comparison of two alternative solutions to support conference calls.

5.1 Related Work

The maintenance of formal specifications has not received the attention it deserves [33]. We seldom see authors modify finished formal specifications for evolving requirements. Students and professionals alike walk away with the impression that formal specification is only suitable for applications with stable requirements. The following is a summary of others' work on specification maintenance.

D. R. Kuhn measures sizes of formal specification updates by *predicate differences*, a notion defined in terms of the effects of variable substitutions on predicates in the specification [98]. There is no guidance on how to update formal specifications for requirement changes.

Khendek and Bochmann create a new labelled transition system (LTS) by merging two finite LTS's. The specifier writes a small LTS for the new behaviour and merges it with the current LTS of the original behaviour [95]. An algorithm can automate the merge if the two LTS's meet certain conditions. Their work allows a large LTS to be built incrementally. The approach is limited to simple events meeting some criteria.

K. J. Turner describes incremental requirements specification of a file system using LOTOS [139]. He called his approach the *constraint-oriented* style. Requirements are specified in a compositional way. A constraint may take the following form resembling a rule in rule-based approaches of specification writing. Unlike our approach, it does not use concrete data in scenarios to aid customer comprehension.

```
if some condition or state applies
   then behaviour is restricted according to the constraint
   else behaviour is unconstrained
```

The idea of *refactoring* was initially used for the improvement of existing code design [53]. It is about the application of small behaviour-preserving transformations to make code more elegant and maintainable. Stepney, Polack and Toyn apply the idea to improve Z specifications [135]. The first of sixteen refactoring transformations they describe is *renaming* which changes an existing name to a more descriptive name. The second transformation is *commonality extraction* which gives a name to a common part appearing in multiple schemas. I am not yet convinced of the merits of refactoring. It may be better to make quality updates to a formal specification than to ruin it with quick and dirty changes just to recover later with refactoring. It is not our aim to dispute here the philosophy of refactoring which is related to agile methodology. Our concern is that refactoring does not update a specification to accommodate requirement changes. Refactoring only makes behaviour-preserving updates to improve the structure of a specification.

5.2 Suppport for Conference Calls

We would like to add the support of conference calls to the basic telephone system with minimal impact to the current usage. The I/O parameters for existing operations should not change. The initiation to join a conference call must come from within a connection. For example, consider that phone users A and B are in a connection but user C is not. Either A or B can phone C. If C picks up the phone, he or she joins the conference. If instead, C calls A or B, a busy ring tone will result.

When requirements change, customers and specifiers work together to capture the new features with scenarios. It may be necessary to revise the data structure. In the previous chapter, a connection is expressed as an ordered pair. This representation is not good for multi-user conferencing. The same conference call can be represented differently using ordered pairs. Following are three of many equivalent representations for a conference of numbers 2, 3, 4, and 5.

$$\{2 \mapsto 3, 3 \mapsto 4, 4 \mapsto 5\}$$
$$\{2 \mapsto 3, 3 \mapsto 4, 3 \mapsto 5\}$$
$$\{2 \mapsto 3, 2 \mapsto 4, 2 \mapsto 5\}$$

It is a chore to determine if two representations are equivalent. The querying and updating of a conference connection on an unsuitable data structure are difficult to express at all levels: scenario, specification and program. Given the first representation above, we learn that numbers 2 and 5 are connected after the inspection of three ordered pairs. When number 4 hangs up, numbers 2, 3, and 5 will remain connected as follows. The update is accomplished by removing an ordered pair and updating another pair. We find the query and update operations tedious to express in the Z notation.

$$\{2 \mapsto 3, 3 \mapsto 5\}$$

5.2.1 Update to State Space

The way a specification is written depends on the data structure it uses. Ordered pairs segregate phone numbers into domain and range. This data structure is unsuitable for the symmetrical relationship of the users in a conference. The problem was less obvious when we only have connections of two numbers. The switch to sets from ordered pairs simplifies the writing of scenarios and the Z specification.

States in English	**Z Expressions**
Nos. $\underline{1}$ and $\underline{4}$ are connected	$\{1, 4\}$
Nos. $\underline{3}$, $\underline{5}$, $\underline{7}$ and $\underline{8}$ are connected	$\{3, 5, 7, 8\}$

Tab. 5.1: Multi-party Connections – from English to Z

PhoneSystem

$connection : \mathbb{P}(\mathbb{P}\, PHONE)$

$ringing : PHONE \nrightarrow PHONE$

State variable *connection* is redefined with the notion of power set. We decide to use the same data structure for normal calls and conferencing. Alternatively, we could keep the old definition of *connection* for 2-party connections and add a new state variable for conferencing. We will compare the two alternatives in the chapter conclusion.

The ordered pair data structure to express *ringing* is unchanged because we still want to distinguish the caller and callee so that the hanging and answering operations can be handled according to the user's role.

5.3 Schema Revisions for New State Space

Existing operation schemas that refer to the modified data structure must be revised. Most operation schemas are affected. The first two predicates of schemas *DialRing* and *DialBusy* on pages 74 and 75 should be changed. The domain and range operators on *connection* are replaced with the generalised union $\bigcup$ which is a unary operator in Z. If s is a set of sets $\{a, b, c, \ \ldots\}$, $\bigcup s$ denotes the smallest set containing all elements that appear in at least one of a, b, c and so on. For example, $\bigcup\{\{1,2\},\{3,4,5\},\{6,7\}\} = \{1,2,3,4,5,6,7\}$.

$$
\begin{array}{|l}
\multicolumn{1}{l}{\mathit{DialRing}} \\ \hline
\Delta \mathit{PhoneSystem} \\
\mathit{caller}?, \mathit{callee}? : \mathit{PHONE} \\
\mathit{tone}! : \mathit{TONE} \\ \hline
\mathit{caller}? \notin \operatorname{dom} \mathit{ringing} \cup \operatorname{ran} \mathit{ringing} \cup \bigcup \mathit{connection} \\
\mathit{callee}? \notin \operatorname{dom} \mathit{ringing} \cup \operatorname{ran} \mathit{ringing} \cup \bigcup \mathit{connection} \\
\mathit{connection}' = \mathit{connection} \\
\mathit{ringing}' = \mathit{ringing} \oplus \{\mathit{caller}? \mapsto \mathit{callee}?\} \\
\mathit{tone}! = \mathit{ring} \\ \hline
\end{array}
$$

$$
\begin{array}{|l}
\multicolumn{1}{l}{\mathit{DialBusy}} \\ \hline
\Xi \mathit{PhoneSystem} \\
\mathit{caller}?, \mathit{callee}? : \mathit{PHONE} \\
\mathit{tone}! : \mathit{TONE} \\ \hline
\mathit{caller}? \notin \operatorname{dom} \mathit{ringing} \cup \operatorname{ran} \mathit{ringing} \cup \bigcup \mathit{connection} \\
\mathit{callee}? \in \operatorname{dom} \mathit{ringing} \cup \operatorname{ran} \mathit{ringing} \cup \bigcup \mathit{connection} \\
\mathit{tone}! = \mathit{busy} \\ \hline
\end{array}
$$

The second predicate in schema *AnswerRing* on page 76 needs revising. Function override $\oplus$ for the maplet is replaced with union $\cup$ for the set. Ordered pair $\mathit{caller} \mapsto \mathit{answer}?$ becomes set $\{\mathit{caller}, \mathit{answer}?\}$.

AnswerRing

$\Delta \mathit{PhoneSystem}$

$\mathit{answer}? : \mathit{PHONE}$

$\mathit{rqt}! : \mathit{RESULT}$

$\mathit{caller} : \mathit{PHONE}$

$\mathit{caller} \mapsto \mathit{answer}? \in \mathit{ringing}$

$\mathit{connection}' = \mathit{connection} \cup \{\{\mathit{caller}, \mathit{answer}?\}\}$

$\mathit{ringing}' = \mathit{ringing} \mathbin{⩥} \{\mathit{answer}?\}$

$\mathit{rqt}! = \mathit{OK}$

Schema *AnswerIgnored* requires no change because it does not refer to variable *connection*.

Schema *HangConnect* is revised so that the first predicate determines the connection *hangSet* that number *hang*? belongs to. The second predicate subtracts this set from *connection*. So far, only schema *AnswerRing* can create a connection. The new connection must be made up of a caller and a callee not already in a connection where the number of the callee is held in variable *answer*?. We can prove that *hangSet* in the following schema *HangConnect* is unique.

HangConnect

$\Delta \mathit{PhoneSystem}$

$\mathit{hang}? : \mathit{PHONE}$

$\mathit{rqt}! : \mathit{RESULT}$

$\mathit{hangSet} : \mathbb{P}\, \mathit{PHONE}$

$\mathit{hang}? \in \mathit{hangSet} \land \mathit{hangSet} \in \mathit{connection}$

$\mathit{connection}' = \mathit{connection} \setminus \{\mathit{hangSet}\}$

$\mathit{ringing}' = \mathit{ringing}$

$\mathit{rqt}! = \mathit{OK}$

The first predicate of schema *HangIgnored* on page 81 is revised with the generalised union operator.

$$
\begin{array}{|l}
\hline
\textit{HangIgnored} \\
\Xi \textit{PhoneSystem} \\
hang? : \textit{PHONE} \\
rqt! : \textit{RESULT} \\
\hline
hang? \notin \operatorname{dom} ringing \cup \bigcup connection \\
rqt! = ignored \\
\hline
\end{array}
$$

Schema *HangRing* needs not be changed.

5.4 A Conference Call Scenario

Scenario *ThreeInConference* shows three numbers in a conference. It begins with numbers 4 and 6 already connected. In state 1, the user at number 6 dials number 2. The phone at number 2 rings. At state 2, the user at number 2 picks up the phone. The phone at number 2 stops ringing. A conference call involving numbers 2, 4 and 6 is established. At state 3, the user at number 6 hangs up. Users at numbers 2 and 4 remain connected. Numbers 7 and 9, staying connected for the whole time, do not play an active role in this scenario.

Step	**Input/Output**	**System State**
0		Nos. 4 and 6 are connected. Nos. 7 and 9 are connected.
1	User at no. 6 dials to no. 2. User gets a ring tone.	Nos. 4 and 6 are connected. Nos. 7 and 9 are connected. No. 6 rings no. 2.
2	User at no. 2 answers the phone. Request is OK.	Nos. 2, 4 and 6 are connected. Nos. 7 and 9 are connected.
3	User at no. 6 hangs up the phone. Request is OK.	Nos. 2 and 4 are connected. Nos. 7 and 9 are connected.

Tab. 5.2: **E-scenario** *ThreeInConference*

Step	**Input**	**Output**	***connection***	***ringing***
0			$\{\{4, 6\}, \{7, 9\}\}$	$\{\ \}$
1	$caller? = 6 \land callee? = 2$	$tone! = ring$	$\{\{4, 6\}, \{7, 9\}\}$	$\{6 \mapsto 2\}$
2	$answer? = 2$	$rqt! = OK$	$\{\{2, 4, 6\}, \{7, 9\}\}$	$\{\ \}$
3	$hang? = 6$	$rqt! = OK$	$\{\{2, 4\}, \{7, 9\}\}$	$\{\ \}$

Tab. 5.3: **Z-scenario** *ThreeInConference*

5.4.1 Verifying a Dialling Step

To see if a schema is capable of handling a step, we substitute the values in the step for the variables in the schema's predicates. If all predicates evaluate to true, it means that the schema can handle the scenario step. We have been doing this through earlier chapters. We shall formally define this useful relationship between schemas and scenarios in the next chapter.

Step *ThreeInConference*.1 in Table 5.3 is a *DiallingOp* based on the names and types of its I/O parameters. After substitutions of the step's data, the first predicate of schema *DialRing* is the only predicate that evaluates to *false*.

$$\begin{aligned}
& \mathit{caller}? \notin \operatorname{dom} \mathit{ringing} \cup \operatorname{ran} \mathit{ringing} \cup \bigcup \mathit{connection} \\
& \Leftrightarrow 6 \notin \operatorname{dom} \emptyset \cup \operatorname{ran} \emptyset \cup \bigcup \{\{4,6\},\{7,9\}\} \\
& \Leftrightarrow 6 \notin \emptyset \cup \emptyset \cup \{4,6,7,9\} \\
& \Leftrightarrow 6 \notin \{4,6,7,9\} \\
& \Leftrightarrow \mathit{false}
\end{aligned}$$

$$\begin{aligned}
& \mathit{callee}? \notin \operatorname{dom} \mathit{ringing} \cup \operatorname{ran} \mathit{ringing} \cup \bigcup \mathit{connection} \\
& \Leftrightarrow 2 \notin \{4,6,7,9\} \\
& \Leftrightarrow \mathit{true}
\end{aligned}$$

$$\begin{aligned}
& \mathit{connection}' = \mathit{connection} \\
& \Leftrightarrow \{\{4,6\},\{7,9\}\} = \{\{4,6\},\{7,9\}\} \\
& \Leftrightarrow \mathit{true}
\end{aligned}$$

$$\begin{aligned}
& \mathit{ringing}' = \mathit{ringing} \oplus \{\mathit{caller}? \mapsto \mathit{callee}?\} \\
& \Leftrightarrow \{6 \mapsto 2\} = \emptyset \oplus \{6 \mapsto 2\} \\
& \Leftrightarrow \{6 \mapsto 2\} = \{6 \mapsto 2\} \\
& \Leftrightarrow \mathit{true}
\end{aligned}$$

$$\begin{aligned}
& \mathit{tone}! = \mathit{ring} \\
& \Leftrightarrow \mathit{ring} = \mathit{ring} \\
& \Leftrightarrow \mathit{true}
\end{aligned}$$

We can modify schema *DialRing* on page 88 by weakening its precondition. The dropping of the generalised union of *connection* makes sense because we want a connected user to be able to ring someone so that a conference call can start or include more phone users.

$$
\begin{array}{|l}
\multicolumn{1}{l}{\textit{DialRing}} \\
\Delta \mathit{PhoneSystem} \\
\mathit{caller}?, \mathit{callee}? : \mathit{PHONE} \\
\mathit{tone}! : \mathit{TONE} \\
\hline
\mathit{caller}? \notin \operatorname{dom} \mathit{ringing} \cup \operatorname{ran} \mathit{ringing} \\
\mathit{callee}? \notin \operatorname{dom} \mathit{ringing} \cup \operatorname{ran} \mathit{ringing} \cup \bigcup \mathit{connection} \\
\mathit{connection}' = \mathit{connection} \\
\mathit{ringing}' = \mathit{ringing} \oplus \{\mathit{caller}? \mapsto \mathit{callee}?\} \\
\mathit{tone}! = \mathit{ring} \\
\hline
\end{array}
$$

5.4.2 Starting a Conference

Schema *AnswerRing* only creates connections of two phone numbers. It does not augment existing connections with additional numbers. We need to create a new schema *AnswerConference* for step *ThreeInConference*.2 in Table 5.3 on page 91.

$$
\begin{array}{|l}
\multicolumn{1}{l}{\textit{AnswerConference}} \\
\Delta \mathit{PhoneSystem} \\
\mathit{answer}? : \mathit{PHONE} \\
\mathit{rqt}! : \mathit{RESULT} \\
\mathit{caller} : \mathit{PHONE} \\
\mathit{callerSet} : \mathbb{P}\, \mathit{PHONE} \\
\hline
\mathit{caller} \mapsto \mathit{answer}? \in \mathit{ringing} \\
\mathit{caller} \in \mathit{callerSet} \land \mathit{callerSet} \in \mathit{connection} \\
\mathit{connection}' = \mathit{connection} \setminus \{\mathit{callerSet}\} \cup \{\mathit{callerSet} \cup \{\mathit{answer}?\}\} \\
\mathit{ringing}' = \mathit{ringing} \rhd \{\mathit{answer}?\} \\
\mathit{rqt}! = \mathit{OK} \\
\hline
\end{array}
$$

There are two local variables in the schema: *caller* and *callerSet*. The first predicate constrains the value of *caller*. The second predicate specifies the value of *callerSet* which holds all the numbers in the connection before the operation. The value of *callerSet* is unique. In other words, a caller cannot be connected to different parties at the same time. Though not shown here, it can be proved by induction based on the way variable *connection* is updated. The third predicate adds the answering number to *callerSet* and substitutes the new set for the original *callerSet* in *connection*. The fourth predicate removes from *ringing* the ordered pair that has the answering number in the range.

We verify schema *AnswerConference* against step *ThreeInConference*.2 by first substituting the values from the step for the variables of the schema. We then try to find suitable values for local variables to make the predicates of the schema true. We can assign value 6 to *caller* to make the first predicate true as shown below.

$$\begin{aligned} & \mathit{caller} \mapsto \mathit{answer?} \in \mathit{ringing} \\ & \Leftrightarrow \mathit{caller} \mapsto 2 \in \{6 \mapsto 2\} \end{aligned}$$

In the evaluation of the second predicate, we can assign value $\{4,6\}$ to *callerSet* to make the predicate true.

$$\begin{aligned} & \mathit{caller} \in \mathit{callerSet} \land \mathit{callerSet} \in \mathit{connection} \\ & \Leftrightarrow 6 \in \mathit{callerSet} \land \mathit{callerSet} \in \{\{4,6\},\{7,9\}\} \end{aligned}$$

The remaining three predicates in the schema evaluate to true as shown below. Schema *AnswerConference* works for step *ThreeInConference*.2.

$$\begin{aligned} & \mathit{connection}' = \mathit{connection} \setminus \{\mathit{callerSet}\} \cup \{\mathit{callerSet} \cup \{\mathit{answer?}\}\} \\ & \Leftrightarrow \{\{2,4,6\},\{7,9\}\} = \{\{4,6\},\{7,9\}\} \setminus \{\{4,6\}\} \cup \{\{4,6\} \cup \{2\}\} \\ & \Leftrightarrow \{\{2,4,6\},\{7,9\}\} = \{\{7,9\}\} \cup \{\{4,6\} \cup \{2\}\} \\ & \Leftrightarrow \{\{2,4,6\},\{7,9\}\} = \{\{7,9\}\} \cup \{\{2,4,6\}\} \\ & \Leftrightarrow \{\{2,4,6\},\{7,9\}\} = \{\{2,4,6\},\{7,9\}\} \\ & \Leftrightarrow \mathit{true} \end{aligned}$$

$$\begin{aligned} & \mathit{ringing}' = \mathit{ringing} \rhd \{\mathit{answer?}\} \\ & \Leftrightarrow \emptyset = \{6 \mapsto 2\} \rhd \{2\} \\ & \Leftrightarrow \emptyset = \emptyset \\ & \Leftrightarrow \mathit{true} \end{aligned}$$

$$\begin{aligned} & \mathit{rqt!} = \mathit{OK} \\ & \Leftrightarrow \mathit{OK} = \mathit{OK} \\ & \Leftrightarrow \mathit{true} \end{aligned}$$

5.4.3 Starting a 2-party Connection

New schema *AnswerConference* and old schema *AnswerRing* overlap in their preconditions. Since the former is handling conference calls for us, we will limit schema *AnswerRing* to create connections with only two users. Shown in the revised version of schema *AnswerRing* on this page, a new predicate ensures that the caller is not a member of an existing connection. Using the definition of the generalised union operator, we could prove that the second predicate in the revised *AnswerRing* and the second predicate of *AnswerConference* are mutually exclusive.

$$\begin{array}{|l}
\multicolumn{1}{l}{\mathit{AnswerRing}} \\ \hline
\Delta \mathit{PhoneSystem} \\
\mathit{answer}? : \mathit{PHONE} \\
\mathit{rqt}! : \mathit{RESULT} \\
\mathit{caller} : \mathit{PHONE} \\ \hline
\mathit{caller} \mapsto \mathit{answer}? \in \mathit{ringing} \\
\mathit{caller} \notin \bigcup \mathit{connection} \\
\mathit{connection}' = \mathit{connection} \cup \{\{\mathit{caller}, \mathit{answer}?\}\} \\
\mathit{ringing}' = \mathit{ringing} \rhd \{\mathit{answer}?\} \\
\mathit{rqt}! = \mathit{OK} \\ \hline
\end{array}$$

We redefine *AnsweringOp* with new *AnswerConference* and revised *AnswerRing*.

$$\mathit{AnsweringOp} \mathrel{\widehat{=}} \mathit{AnswerRing} \lor \mathit{AnswerConference} \lor \mathit{AnswerIgnored}$$

5.4.4 Ending a Conference

Step *ThreeInConference*.2 in Table 5.3 on page 91 shows a user hanging up. The remaining users are still connected. To determine the new value of *connection*, we first find the set that holds the hanging number and subtract the set from the set of connections. We then add back the connection without the hanging number. As before, *hangSet* is unique.

$$\mathit{HangConference}$$

$$\begin{array}{l}
\Delta \mathit{PhoneSystem} \\
\mathit{hang}? : \mathit{PHONE} \\
\mathit{rqt}! : \mathit{RESULT} \\
\mathit{hangSet} : \mathbb{P}\, \mathit{PHONE} \\
\hline
\mathit{hang}? \in \mathit{hangSet} \land \mathit{hangSet} \in \mathit{connection} \\
\# \mathit{hangSet} > 2 \\
\mathit{connection}' = \mathit{connection} \setminus \{\mathit{hangSet}\} \cup \{\mathit{hangSet} \setminus \{\mathit{hang}?\}\} \\
\mathit{ringing}' = \mathit{ringing} \\
\mathit{rqt}! = \mathit{OK}
\end{array}$$

We verify schema *HangConference* against step *ThreeInConference*.3 as follows. In the evaluation of the first predicate, we can only assign value $\{2, 4, 6\}$ to local variable *hangSet* to make the predicate true.

$$\begin{array}{l}
\mathit{hang}? \in \mathit{hangSet} \land \mathit{hangSet} \in \mathit{connection} \\
\Leftrightarrow 6 \in \mathit{hangSet} \land \mathit{hangSet} \in \{\{2, 4, 6\}, \{7, 9\}\}
\end{array}$$

All remaining predicates in *HangConference* evaluate to true. Having the size of *hangSet* greater than 2 ensures that the schema is only applied to a conference call.

$\#hangSet > 2$
$\Leftrightarrow \#\{2,4,6\} > 2$
$\Leftrightarrow 3 > 2$
$\Leftrightarrow true$

$connection' = connection \setminus \{hangSet\} \cup \{hangSet \setminus \{hang?\}\}$
$\Leftrightarrow \{\{2,4\},\{7,9\}\} = \{\{2,4,6\},\{7,9\}\} \setminus \{\{2,4,6\}\} \cup \{\{2,4,6\} \setminus \{6\}\}$
$\Leftrightarrow \{\{2,4\},\{7,9\}\} = \{\{7,9\}\} \cup \{\{2,4,6\} \setminus \{6\}\}$
$\Leftrightarrow \{\{2,4\},\{7,9\}\} = \{\{7,9\}\} \cup \{\{2,4\}\}$
$\Leftrightarrow \{\{2,4\},\{7,9\}\} = \{\{2,4\},\{7,9\}\}$
$\Leftrightarrow true$

$ringing' = ringing$
$\Leftrightarrow \emptyset = \emptyset$
$\Leftrightarrow true$

$rqt! = OK$
$\Leftrightarrow OK = OK$
$\Leftrightarrow true$

5.4.5 Ending a 2-party Connection

Schema *HangConnect* was written with the assumption that all connections have only two users. It did not check this condition explicitly. Now we need to strengthen its precondition with the second predicate below. Without the checking of set size, schema *HangConnect* will interfere with the operation of schema *HangConference*.

$$
\begin{array}{|l}
\multicolumn{1}{l}{\mathit{HangConnect}} \\
\hline
\Delta \mathit{PhoneSystem} \\
\mathit{hang}? : \mathit{PHONE} \\
\mathit{rqt}! : \mathit{RESULT} \\
\mathit{hangSet} : \mathbb{P}\, \mathit{PHONE} \\
\hline
\mathit{hang}? \in \mathit{hangSet} \land \mathit{hangSet} \in \mathit{connection} \\
\neg\ (\#\mathit{hangSet} > 2) \\
\mathit{connection}' = \mathit{connection} \setminus \{\mathit{hangSet}\} \\
\mathit{ringing}' = \mathit{ringing} \\
\mathit{rqt}! = \mathit{OK} \\
\hline
\end{array}
$$

HangingOp is redefined with new schema *HangConference*.

$$\mathit{HangingOp} \mathrel{\widehat{=}} \mathit{HangConnect} \lor \mathit{HangRing} \lor \mathit{HangConference} \lor \mathit{HangIgnored}$$

5.5 Reverification

For every scenario step, the formal specification needs to have an operation to handle it. An operation is defined as a disjunction of schemas. Every time there are new or revised scenarios, after making necessary modifications to the specification, we should reverify it. If we have a tool that keeps track of scenario steps and their handling schemas, the reverification can be done selectively on the affected scenarios and schemas.

We should consider to modify an existing schema before adding a new one. It can prevent the number of schemas from growing unnecessarily. We only need to consider the schemas used to define the same operation. For example, if a new scenario step has I/O parameters matching those of *OP*, we just need to see if any of *Disjunct1*, *Disjunct2* and *Disjunct3* can handle the step with or without modification.

$$OP \mathrel{\widehat=} \mathit{Disjunct}1 \lor \mathit{Disjunct}2 \lor \mathit{Disjunct}3$$

It is possible for customers to drop the requirements captured in some scenarios. Reverification identifies the schemas not used for any steps. We can eliminate the unused schemas from the formal specification. Suppose *Disjunct*3 is not used anymore, we can redefine *OP*.

$$OP \mathrel{\widehat=} \mathit{Disjunct}1 \lor \mathit{Disjunct}2$$

After modifications on the operations, we shall check that the operations are still total and deterministic. For example, the deletion of *Disjunct3* may create a hole in the precondition of *OP*.

Finally, customers can periodically review the scenarios to see that they are still current and complete.

5.6 Chapter Conclusion

The majority of formal method authors have unrealistically treated customer requirements as being static. The existing work has severe limitations. Some only deal with event-based specifications. Refactoring only deals with the clean-up after quick and dirty changes but not how to put the changes in the specification.

In the previous chapter, we have made a bad decision on the data structure used to represent connections. Ordered pairs are only good for two numbers in a connection. Much work is involved in changing data structures. We have to revise all existing schemas for the new data structure even before we can consider conferencing scenarios. Only a foreknowledge of future requirements can safeguard us from the predicament.

We chose to represent a conference of multiple users by a set. We picked the same data structure for 2-party connections and conference calls. This decision may allow us to use the same schema whether an operation relates to a normal connection or a conference.

We could have made a different decision to keep ordered pairs for normal connections and only use sets for conference calls. This alternative choice would eliminate the schema revision work we did in section 5.3. However the short term gain may be more expensive in the long run because we will always need to keep two schemas of each operation for 2-party and multi-party connections. To query the status of a line, we will also need to check two state variables. Therefore we decided to bite the bullet and convert all connections to sets.

Following is a number of tasks that specifiers would try in order for new scenarios.

1. Try existing schemas
2. Try to modify existing schemas
3. Create new schemas on existing state space
4. Revise state space and create new schemas

6. SPECIFICATION VERIFICATION

We have introduced an approach to create operation schemas by generalising concrete values in scenarios to variables. Relationships of the variables are captured in schema predicates. Given our limited cognitive capacity, we can only attend to a few scenario steps when writing an operation schema. For example, in Chapter 4 we created schemas by considering two or three complementary scenario steps together. To gain more confidence of their completeness and correctness, we verify schemas against additional scenario steps. This chapter presents the underpinning of schema verification against scenario steps that we have been doing in earlier chapters.

In the formal definition of a programming language, we could define the language syntax, the semantic domain and the function that maps the syntax to its semantic value [143, Chapter 4]. That approach of defining a programming language does not work well for our formalising effort of concrete scenarios. It presumes that any particular program gives rise to a unique meaning. Our authoring of operation schemas from scenarios on the other hand is creative. Design decisions are made by the developer in the process. A set of concrete scenarios does not map to a unique set of schemas.

We specify the formal relationship between a set of concrete scenarios and a collection of operation schemas with *scenarioObserved* a Boolean function. It returns *true* when the concrete scenarios in its first parameter are observed by the schemas in its second parameter. With different developer's decisions, we will derive different schemas from the same concrete scenarios. The function will still return *true*. The function is built on top of a simpler and similar function called *stepObserved* that returns *true* when the scenario step in its first parameter is realised by an implementation of the schema in its second parameter.

Section 6.1 defines the syntax of variable names and scenario steps. Section 6.2 defines basic functions. Function *pre2post* maps pre-state variable names to corresponding post-state variable names. Several other functions return the names of various kinds of variables used in an operation schema. Section 6.3 defines concrete scenarios concluding the treatment of syntax. We then turn to semantics and define step observance in Section 6.4. The observance relation is extended from steps to scenarios in Section 6.5. The examples in this chapter are taken from the telephone system problem.

6.1 Basic Types

We borrow symbols from Z notation for our formal definition. The basic types defined in this section do not change even when we move to different problem domains.

6.1.1 Variable Names

X-TYPE, *Y-TYPE*, *I-TYPE* and *O-TYPE* respectively hold the names of all pre-state, post-state, input and output variables. The four types are used in scenarios and schemas.

Definition 6.1.1.

$$\begin{aligned} &\textit{X-TYPE} &&== C^{+} \\ &\textit{Y-TYPE} &&== C^{+\prime} \\ &\textit{I-TYPE} &&== C^{+}? \\ &\textit{O-TYPE} &&== C^{+}! \end{aligned}$$

where C stands for an alphanumeric character,
$^{+}$ *is a metasymbol for non-zero repetitions of the construct before it,*
and ′, ? and ! are terminal symbols.

Definition 6.1.2.

$$\textit{L-TYPE} == C^{+}$$

L-TYPE holds the names of all local variables. L-TYPE is not used in scenarios because they do not contain local variables. Both pre-state and local variable names are strings of alphanumeric characters. Developers need to avoid the conflicting use of the same name as a pre-state variable and a local variable in the same operation schema.

Example

$$
\begin{aligned}
&\mathit{connection} \in \mathit{X\text{-}TYPE}\\
&\mathit{connection}' \in \mathit{Y\text{-}TYPE}\\
&\mathit{hang}? \in \mathit{I\text{-}TYPE}\\
&\mathit{rqt}! \in \mathit{O\text{-}TYPE}\\
&\mathit{hangSet} \in \mathit{L\text{-}TYPE}
\end{aligned}
$$

6.1.2 Values

V-TYPE is the type for values that can be stored in the variables. Integers and strings are just some possible values allowed by *V-TYPE*. We use an inclusive definition below to allow any values.

Definition 6.1.3. *Set V-TYPE contains all permissible values that may be assigned to variables.*

6.1.3 Scenario Steps

STEP-TYPE is the type for scenario steps. A step is a partial function which maps pre-state, post-state, input and output variables to values.

Definition 6.1.4.

$$\mathit{STEP\text{-}TYPE} == (\mathit{X\text{-}TYPE} \cup \mathit{Y\text{-}TYPE} \cup \mathit{I\text{-}TYPE} \cup \mathit{O\text{-}TYPE}) \nrightarrow \mathit{V\text{-}TYPE}$$

Example

The first step of scenario *ThreeInConference* on page 91 has the following value from STEP-TYPE.

$$
\begin{aligned}
\{\ &\mathit{connection} \mapsto \{\{4,6\},\{7,9\}\},\\
&\mathit{ringing} \mapsto \emptyset,\\
&\mathit{connection}' \mapsto \{\{4,6\},\{7,9\}\},\\
&\mathit{ringing}' \mapsto \{6 \mapsto 2\},\\
&\mathit{caller}? \mapsto 6, \mathit{callee}? \mapsto 2,\\
&\mathit{tone}! \mapsto \mathit{ring}\}
\end{aligned}
$$

6.2 Basic Functions

6.2.1 Mapping Variables from Pre-state to Post-state

Function *pre2post* maps a pre-state variable name to its post-state variable name by appending the terminal symbol prime $'$ to the end. The function helps us express the continuity of the steps in a scenario.

Definition 6.2.1.

$pre2post : X\text{-}TYPE \rightarrow Y\text{-}TYPE$
$pre2post = \{x : X\text{-}TYPE \bullet x'\}$

Example

connection is a pre-state variable and *connection′* is the corresponding post-state variable. This fact is represented by the following relation.

$$connection \mapsto connection' \in pre2post$$

6.2.2 Variables Used By Schemas

The types and function defined earlier are universal. They do not change when we move from one application domain to the next. From this point on, however, all definitions are tailored to individual application domains.

Operation schemas consist of declarations and predicates. Declarations name the variables that the predicates use. We define a few functions to return the various kinds of variables declared in a schema.

Definition 6.2.2. *Functions pre(h), post(h), in(h) and out(h) respectively contain the names of the global pre-state, global post-state, input and output variables used in an operation schema h.*

$pre(h) \subset X\text{-}TYPE$
$post(h) \subset Y\text{-}TYPE$
$in(h) \subset I\text{-}TYPE$
$out(h) \subset O\text{-}TYPE$

Example

Schema *HangConnect* on page 98 declares *PhoneSystem*, *hang*? and *rqt*!. *PhoneSystem* in turn consists of *connection* and *ringing*.

$$\begin{aligned}
pre(h) &= \{connection, ringing\}\\
post(h) &= \{connection', ringing'\}\\
in(h) &= \{hang?\}\\
out(h) &= \{rqt!\}
\end{aligned}$$

6.3 Concrete Scenarios

Before formally defining concrete scenarios, we illustrate domain restriction operator $\lhd$ and function composition operator $\fatsemi$ from Z with examples.

Example

We place *X-TYPE* and *Y-TYPE* to the left of domain restriction operator $\lhd$ to extract the mappings of pre-state and post-state variables respectively.

$$\begin{aligned}
X\text{-}TYPE \lhd \{\ & connection \mapsto \{\{4,6\},\{7,9\}\}, ringing \mapsto \emptyset,\\
& connection' \mapsto \{\{4,6\},\{7,9\}\}, ringing' \mapsto \{6 \mapsto 2\},\\
& caller? \mapsto 6, callee? \mapsto 2, tone! \mapsto ring\}\\
= \{connection \mapsto & \{\{4,6\},\{7,9\}\}, ringing \mapsto \emptyset\}
\end{aligned}$$

$$\begin{aligned}
Y\text{-}TYPE \lhd \{\ & connection \mapsto \{\{4,6\},\{7,9\}\}, ringing \mapsto \emptyset,\\
& connection' \mapsto \{\{4,6\},\{7,9\}\}, ringing' \mapsto \{6 \mapsto 2\},\\
& caller? \mapsto 6, callee? \mapsto 2, tone! \mapsto ring\}\\
= \{connection' \mapsto & \{\{4,6\},\{7,9\}\}, ringing' \mapsto \{6 \mapsto 2\}\}
\end{aligned}$$

Example

To the left of operator $\fatsemi$, we have a set that maps pre-state variable *ringing* to post-state variable *ringing'*. Composing it with a mapping from a post-state variable to a value, we get a mapping from the corresponding pre-state variable to the same value.

$$
\begin{aligned}
&\{ringing \mapsto ringing'\} \mathbin{;} \{ringing' \mapsto \{6 \mapsto 2\}\}\\
&= \{ringing \mapsto \{6 \mapsto 2\}\}
\end{aligned}
$$

Definition 6.3.1.

$$
\begin{aligned}
&\mathit{SCENARIO\text{-}TYPE} ==\\
&\qquad \{f : \mathbb{N} \mathrel{+\!\!\!\!\twoheadrightarrow} \mathit{STEP\text{-}TYPE} \mid \operatorname{dom} f = 1 \mathinner{..} \# f \wedge\\
&\qquad\qquad (\,\forall\, i \in 1 \mathinner{..} (\# f - 1) \bullet\\
&\qquad\qquad\quad (pre2post \mathbin{;}\ \mathit{Y\text{-}TYPE} \lhd f(i)) = (\ \mathit{X\text{-}TYPE} \lhd f(i+1))))\}
\end{aligned}
$$

A concrete scenario is a finite partial function indicated by Z symbol $\mathrel{+\!\!\!\!\twoheadrightarrow}$. The first conjunct describes that the domain of the function consists of consecutive natural numbers. The range of the function are scenario steps. A scenario step maps pre-state, post-state, input and output variables to values. The second conjunct requires the post-state of a step to match the pre-state of the following step. Expression $\mathit{Y\text{-}TYPE} \lhd f(i)$ restricts the domain of step $f(i)$ to post-state variables.

When writing a formal specification, if the value of a variable remains unchanged by an operation, we can write that a variable holding a range of values in one state implies the same variable holding a larger range of values in the next state, for example $(0 \leqslant i \leqslant 5) \Rightarrow (0 \leqslant i' \leqslant 9)$. But concrete scenarios deal with exact value assignments instead of value ranges. We can afford to be more precise to write that after composing with *pre2post* the post-state of a step equals the pre-state of the next step.

In earlier chapters, we attached alphabetic subscripts to some data values for documentation and understanding. They are not a part of the formal definition.

Example

Given individual scenario steps t_1, t_2 and t_3 as follows:

$$
\begin{aligned}
t_1 = \{\ & connection \mapsto \{\{4,6\},\{7,9\}\},\\
& ringing \mapsto \emptyset,\\
& connection' \mapsto \{\{4,6\},\{7,9\}\},\\
& ringing' \mapsto \{6 \mapsto 2\},\\
& caller? \mapsto 6, callee? \mapsto 2,
\end{aligned}
$$

$$\mathit{tone}! \mapsto \mathit{ring}\}$$

$$\begin{aligned} t_2 = \{\ & \mathit{connection} \mapsto \{\{4,6\},\{7,9\}\}, \\ & \mathit{ringing} \mapsto \{6 \mapsto 2\}, \\ & \mathit{connection}' \mapsto \{\{2,4,6\},\{7,9\}\}, \\ & \mathit{ringing}' \mapsto \emptyset, \\ & \mathit{answer}? \mapsto 2, \\ & \mathit{rqt}! \mapsto \mathit{OK}\} \end{aligned}$$

$$\begin{aligned} t_3 = \{\ & \mathit{connection} \mapsto \{\{2,4,6\},\{7,9\}\}, \\ & \mathit{ringing} \mapsto \emptyset, \\ & \mathit{connection}' \mapsto \{\{2,4\},\{7,9\}\}, \\ & \mathit{ringing}' \mapsto \emptyset, \\ & \mathit{hang}? \mapsto 6, \\ & \mathit{rqt}! \mapsto \mathit{OK}\} \end{aligned}$$

Scenario *ThreeInConference* originally in a table form on page 91 can be expressed as a set $\{1 \mapsto t_1, 2 \mapsto t_2, 3 \mapsto t_3\}$ in agreement to Definition 6.3.1. The domain of a concrete scenario is a set of consecutive numbers. In Z, such a set of mappings can be written compactly as a sequence $\langle t_1, t_2, t_3\rangle$.

The first conjunct in the concrete scenario definition requires all steps to be of *STEP-TYPE.* The proof is so trivial that it is hardly necessary. We just need to show that every step is a partial function with domain *X-TYPE* ∪ *Y-TYPE* ∪ *I-TYPE* ∪ *O-TYPE* and range *V-TYPE.*

The second conjunct requires a step's post-state to match the next step's pre-state. There are $n-1$ matches to prove in an n-step scenario. Consider $i = 1$ for the first of the two required matches in our 3-step scenario.

$$\begin{aligned} t_1 = \{\ & \mathit{connection} \mapsto \{\{4,6\},\{7,9\}\}, \\ & \mathit{ringing} \mapsto \emptyset, \\ & \mathit{connection}' \mapsto \{\{4,6\},\{7,9\}\}, \\ & \mathit{ringing}' \mapsto \{6 \mapsto 2\}, \\ & \mathit{caller}? \mapsto 6, \mathit{callee}? \mapsto 2, \\ & \mathit{tone}! \mapsto \mathit{ring}\} \end{aligned}$$

$$\text{Y-TYPE} \lhd t_1 = \{\mathit{connection}' \mapsto \{\{4,6\},\{7,9\}\}, \mathit{ringing}' \mapsto \{6 \mapsto 2\}\}$$

$$\begin{aligned} & \mathit{pre2post} \mathbin{⨾} \text{Y-TYPE} \lhd t_1 \\ & = \{\mathit{connection} \mapsto \mathit{connection}', \mathit{ringing} \mapsto \mathit{ringing}', \ldots\} \mathbin{⨾} \\ & \qquad\qquad \{\mathit{connection}' \mapsto \{\{4,6\},\{7,9\}\}, \mathit{ringing}' \mapsto \{6 \mapsto 2\}\} \\ & = \{\mathit{connection} \mapsto \{\{4,6\},\{7,9\}\}, \mathit{ringing} \mapsto \{6 \mapsto 2\}\} \end{aligned}$$

$$\begin{aligned} t_2 = \{ \; & \mathit{connection} \mapsto \{\{4,6\},\{7,9\}\}, \\ & \mathit{ringing} \mapsto \{6 \mapsto 2\}, \\ & \mathit{connection}' \mapsto \{\{2,4,6\},\{7,9\}\}, \\ & \mathit{ringing}' \mapsto \emptyset, \\ & \mathit{answer}? \mapsto 2, \\ & \mathit{rqt}! \mapsto \mathit{OK}\} \end{aligned}$$

$$\text{X-TYPE} \lhd t_2 = \{\mathit{connection} \mapsto \{\{4,6\},\{7,9\}\}, \mathit{ringing} \mapsto \{6 \mapsto 2\}\}$$

$$\therefore \;\; \mathit{pre2post} \mathbin{⨾} \text{Y-TYPE} \lhd t_1 = \text{X-TYPE} \lhd t_2$$

$$\therefore 1 < 3 \Rightarrow (\mathit{pre2post} \mathbin{⨾} \text{Y-TYPE} \lhd \; t_1) = (\text{ X-TYPE} \lhd \; t_2)$$

The consideration of the case for $i = 2$ is required to complete the proof of the second conjunct in Definition 6.3.1 on page 106. We will skip it because of its similarity to the case for $i = 1$ which we have just proved. When a scenario is expressed with a table, a step is actually represented by two consecutive rows. In Table 5.3 on page 91, rows 0 and 1 are the pre-state and post-state of step t_1. Rows 1 and 2 are the pre-state and post-state of step t_2. Rows 2 and 3 are the pre-state and post-state of step t_3. The representation guarantees the matching of the post-state of one step with the pre-state of the next step. For example, row 2 is simultaneously the post-state of t_2 and the pre-state of t_3.

6.4 Step Observance

We use square brackets to represent the substitutions of values for variables. For example, $h[t]$ denotes the evaluation of schema h after the variables in its predicates are substituted with the values in step t. Schema h may have additional local variables not specified in the step. Function u specifies suitable values for the local variables declared in schema h. If h evaluates to true after the appropriate substitutions for step t and the local variables, schema h is said to observe step t.

Definition 6.4.1.

$$\begin{aligned}
&\mathit{stepObserved} : \mathit{STEP\text{-}TYPE} \times \mathit{SCHEMA} \to \mathit{BOOLEAN}\\
&\mathit{stepObserved} = \{t : \mathit{STEP\text{-}TYPE}\ ;\ h : \mathit{SCHEMA} \mid\\
&\qquad\qquad \mathit{X\text{-}TYPE} \lhd t = \mathit{pre}(h) \land\\
&\qquad\qquad \mathit{Y\text{-}TYPE} \lhd t = \mathit{post}(h) \land\\
&\qquad\qquad \mathit{I\text{-}TYPE} \lhd t = \mathit{in}(h) \land\\
&\qquad\qquad \mathit{O\text{-}TYPE} \lhd t = \mathit{out}(h) \land\\
&\qquad\qquad \exists\, u \in (\mathit{L\text{-}TYPE} \nrightarrow \mathit{V\text{-}TYPE}) \bullet h[t][u]\}
\end{aligned}$$

The first four conjuncts in the definition ensure that the scenario step and the schema cover the same state, input and output variables.

Example

We would like to prove that h *observes* t for h = schema *HangConference* on page 96 and t = *ThreeInConference*.3 which is step t_3 on page 107.

$$\begin{aligned}
&h[t]\\
&\Leftrightarrow \mathit{hang?} \in \mathit{hangSet} \land \mathit{hangSet} \in \mathit{connection} \land \#\mathit{hangSet} > 2 \land\\
&\quad \mathit{connection}' = \mathit{connection} \setminus \{\mathit{hangSet}\} \cup \{\mathit{hangSet} \setminus \{\mathit{hang?}\}\} \land\\
&\quad \mathit{ringing}' = \mathit{ringing} \land \mathit{rqt!} = \mathit{OK}\ [t]\\
&\Leftrightarrow 6 \in \mathit{hangSet} \land \mathit{hangSet} \in \{\{2,4,6\},\{7,9\}\} \land \#\mathit{hangSet} > 2 \land\\
&\quad \{\{2,4\},\{7,9\}\} = \{\{2,4,6\},\{7,9\}\} \setminus \{\mathit{hangSet}\} \cup \{\mathit{hangSet} \setminus \{6\}\} \land\\
&\quad \emptyset = \emptyset \land \mathit{OK} = \mathit{OK}\\
&\Leftrightarrow 6 \in \mathit{hangSet} \land \mathit{hangSet} \in \{\{2,4,6\},\{7,9\}\} \land \#\mathit{hangSet} > 2 \land\\
&\quad \{\{2,4\},\{7,9\}\} = \{\{2,4,6\},\{7,9\}\} \setminus \{\mathit{hangSet}\} \cup \{\mathit{hangSet} \setminus \{6\}\}
\end{aligned}$$

We need to find a suitable function u so that $h[t][u]$ evaluates to true. Local variable *hangSet* is constrained by the second conjunct above to two possible values $\{2,4,6\}$ and $\{7,9\}$. We will try $u = \{\mathit{hangSet} \mapsto \{2,4,6\}\}$ first.

$h[t][u]$

$\Leftrightarrow 6 \in \mathit{hangSet} \wedge \mathit{hangSet} \in \{\{2,4,6\},\{7,9\}\} \wedge \#\mathit{hangSet} > 2 \wedge$

$\quad \{\{2,4\},\{7,9\}\} = \{\{2,4,6\},\{7,9\}\} \setminus \{\mathit{hangSet}\} \cup \{\mathit{hangSet} \setminus \{6\}\}$

$\quad [\ \{\mathit{hangSet} \mapsto \{2,4,6\}\}\ ]$

$\Leftrightarrow 6 \in \{2,4,6\} \wedge \{2,4,6\} \in \{\{2,4,6\},\{7,9\}\} \wedge \#\{2,4,6\} > 2 \wedge$

$\quad \{\{2,4\},\{7,9\}\} = \{\{2,4,6\},\{7,9\}\} \setminus \{\{2,4,6\}\} \cup \{\{2,4,6\} \setminus \{6\}\}$

$\Leftrightarrow \mathit{true} \wedge \mathit{true} \wedge 3 > 2 \wedge$

$\quad \{\{2,4\},\{7,9\}\} = \{\{7,9\}\} \cup \{\{2,4,6\} \setminus \{6\}\}$

$\Leftrightarrow \mathit{true} \wedge \{\{2,4\},\{7,9\}\} = \{\{7,9\},\{2,4\}\}$

$\Leftrightarrow \mathit{true}$

Schema *hangConference* observes step *ThreeInConference*.3.

6.5 Scenario Observance

We define an operation as the disjunction of a number of schemas sharing the same input and output parameters. The definition reflects our view of programs described in Section 1.7. As explained, given the flexibility allowed within the predicates of a schema, this view does not compromise the generality of the programs we can express.

Definition 6.5.1.

$\mathit{OPERATION} == h_1 \vee h_2 \vee \ \ldots \ \vee h_n$

$\mathit{where}\ n \in \mathbb{N} \wedge$

$\qquad \forall\, i,j \in 1 \mathrel{..} n \bullet h_i \in \mathit{SCHEMA} \wedge \mathit{in}(h_i) = \mathit{in}(h_j) \wedge \mathit{out}(h_i) = \mathit{out}(h_j)$

A set of operations observes a set of concrete scenarios if and only if for every step t in the scenarios, there exists an operation h such that h observes t. This relationship is captured by Boolean function *scenarioObserved*.

Definition 6.5.2.

$\mathit{scenarioObserved} : \mathbb{P}\ \mathit{SCENARIO\text{-}TYPE} \times \mathbb{P}\ \mathit{OPERATION} \rightarrow \mathit{BOOLEAN}$

$\mathit{scenarioObserved} = \{\, Q : \mathbb{P}\ \mathit{SCENARIO\text{-}TYPE}\ ;\ H : \mathbb{P}\ \mathit{OPERATION} \mid$

$\qquad (\forall\, q \in Q;\ t \in q;\ \exists\, h \in H\ \bullet \mathit{stepObserved}(t,h))\}$

6.6 Chapter Conclusion

We have defined four basic types X-TYPE, Y-TYPE, I-TYPE and O-TYPE for variable names and another basic type V-TYPE for all possible values. Post-state variables in X-TYPE differ from their corresponding pre-state variables by a trailing prime. The basic types are universal. They do not change when we move from one problem domain to another.

Step type STEP-TYPE is a function that maps basic variable types to value type V-TYPE. A concrete scenario is a sequence of consecutive steps where the post-state variables of one step must match the pre-state variables of the next step.

An operation schema observes a scenario step if the schema predicates evaluate to true after suitable value substitutions for variables. The observance relation is also defined between a set of schemas and a set of scenarios. A set of operation schemas observes a set of concrete scenarios if there exists a schema to observe every scenario step.

The observance of a scenario step by a schema only means that the schema does not contradict the step. Observance alone does not preclude another step from having the same input and pre-state but difference output and post-state. Observance has to be combined with determinism to avoid undesirable outcomes. Determinism will be discussed in Chapter 8.

7. SCENARIO EXPANSIONS VERSUS SPECIFICATION REFINEMENT

The concrete scenarios in the previous chapters are written at an abstraction level for the customers. They are **customer scenarios**. Their steps are called **customer steps** describing only what need to be done. All variables used concern the customer. Otherwise they would not be part of a customer scenario.

The developer may expand a customer step into multiple **developer steps** to describe how it can be accomplished. The expanded scenarios are also called **developer scenarios**. The expansions embody choices of algorithms. The expanded steps are described with new states and variables which do not concern the customer. Developers can add details to scenarios useful for implementations.

In this chapter, we work on the sorting problem. The customer sees it as a single operation. For the developer, a single sorting operation is accomplished with many swaps passing through a number of intermediate states. If we allow the swaps to take place in any order, an initial state can reach its final state through different sequences of intermediate states. These expanded scenarios are generalised to a specification. If the path from an initial state to its final state is fixed by an algorithm, the expanded scenarios are generalised to a program instead.

Customers use scenarios to express and document their requirements. Developers use expanded scenarios to visualise and document the detailed steps needed to accomplish user tasks. Customer and developer scenarios both facilitate communications between project team members.

Specifiers generalise customer scenarios to create specifications. It is common for formal method practitioners to refine specifications to programs. Alternatively, programmers may expand the customer scenarios and generalise the expanded scenarios to programs. The programs can be verified directly against scenarios without using formal specifications.

7.1 Sorting Specification

We have the task to sort a number of records. To focus on the key concepts, we use integers as record keys and ignore other record components.

7.1.1 First Order Logic

Following is a formal specification adapted from [39, page 318]. Symbols $\frown$ and $\uplus$ stand for list concatenation and bag union respectively. Lists are enclosed in $\langle\ \rangle$ and bags in $[\![\ ]\!]$.

$$\begin{aligned}
&\mathit{Sort} : \mathbb{N}^* \rightarrow \mathbb{N}^*\\
&\text{pre-}\mathit{Sort}(l) \mathrel{\widehat{=}} \text{True}\\
&\text{post-}\mathit{Sort}(l_u, l_s) \mathrel{\widehat{=}} \text{bag}(l_u) = \text{bag}(l_s) \wedge \mathit{ascending}(l_s)
\end{aligned}$$

where

$$\begin{aligned}
&\text{bag}(\langle\rangle) \mathrel{\widehat{=}} \emptyset\\
&\text{bag}(\langle x\rangle) \mathrel{\widehat{=}} [\![x]\!]\\
&\text{bag}(l_1 \frown l_2) \mathrel{\widehat{=}} \text{bag}(l_1) \uplus \text{bag}(l_2)
\end{aligned}$$

$$\begin{aligned}
&\mathit{ascending}(l) \mathrel{\widehat{=}} \forall\, x, y : \mathbb{N} \bullet \mathit{inOrder}(x, y, l) \Rightarrow x \leq y\\
&\mathit{inOrder}(x, y, l) \mathrel{\widehat{=}} \exists\, l_1, l_2, l_3 : \mathbb{N}^* \bullet l = l_1 \frown \langle x\rangle \frown l_2 \frown \langle y\rangle \frown l_3
\end{aligned}$$

The postcondition specified in post-$\mathit{Sort}(l_u, l_s)$ has two conjuncts. Function *bag* turns a list into a bag which keeps the count of each key but ignores the order of the keys in the list. The first conjunct $\text{bag}(l_u) = \text{bag}(l_s)$ states that the counts of each key in unsorted list l_u and sorted list l_s are equal. The second conjunct $\mathit{ascending}(l_s)$ means that the keys in the output list must be in ascending order.

7.1.2 Z

The above specification in first order logic can be written in Z notation as follows where *items* is the Z function to turn a sequence into a bag. Sequences and lists are different names for the same structure. The first predicate ensures that both lists contain the same keys, and if applicable the same number of duplications. In Z, we use the bracket notation to refer to individual squence members. For example, $ls?(i)$ refers to the ith member of input sequence $ls?$. The second predicate ensures that the keys in the sorted list are in ascending order. The two predicates here correspond to the two conjuncts of the specification in first order logic above.

SortSpec

$lu?, ls? : \operatorname{seq} \mathbb{N}$

$items\ lu? = items\ ls?$

$\forall i, j : 1 \mathrel{..} 50 \bullet i < j \Rightarrow ls?(i) \leq ls?(j)$

Like all other Z schemas that appear in the thesis, the above schema has been verified to comply with Z syntax using type checker ZTC [91]. We also use a Z animator called ZANS to execute schemas [89]. ZANS lacks the capability of programming languages like Prolog to automatically find the correct values of sorted list *ls* for unsorted list *lu*. To animate the schema, we need to code the sorted list as an input parameter indicated by the trailing question mark.

We have hard coded the above schema to sort a list of 50 keys. ZANS only animates a subset of Z [90] and it does not evaluate size operator # correctly within a *for all* construct. Should we change the hard coded size 50 to the more general expression $\#ls?$ for arbitrary list size, ZANS returns an empty range and mistakenly considers the *for all* predicate true even when the keys in *ls* are unsorted. For the other schemas in the thesis, we present the normal version that passed the type checker rather than the idiosyncratic version adapted for execution on the animator.

This is about as simple as a useful formal specification can get. However a few years ago when I wrote the sorting specification, I forgot the permutation requirement. It is difficult to tell when we miss a part of the complete specification. Another problem is that customers would not be able to tell us if a formal specification is correct since they do not understand the notations used. This is why we turn to this scenario-driven approach to create a specification or program.

7.2 Scenario-Based Specification

We normally begin by writing domain concepts in sentences using a natural language. We confine the sentences to a small number of templates for manageability. The sentences are then translated to Z expressions. Since sorting is a familiar problem, we shall skip the sentences and jump right into Z-scenarios. We consider a scenario with distinct keys and another scenario with duplicated keys.

7.2.1 Distinct Key Scenario

There are no user interactions necessary. In addition to a column for state variable *KeyList*, we use a new column *condition* to capture the significant data relationships in the respective state.

Step	***keyList***	**Condition**
0	$\langle 8, 4, 2, 6 \rangle$	
1	$\langle 2, 4, 6, 8 \rangle$	$2 \leq 4 \leq 6 \leq 8$

Tab. 7.1: **Z-scenario** *SortFourKeys*

State 1 of the scenario in Table 7.1 has the terminating condition of $2 \leq 4 \leq 6 \leq 8$ for a sorted *KeyList*. We prefer this contracted syntax over the lengthened version $2 \leq 4 \land 4 \leq 6 \land 6 \leq 8$ which favours compilers over human readers.

From the scenario, we observe that three of the four positions in the finishing list have their values altered. We use a basic operation that swaps two values. The change of three values is effected by two or more swaps. The one-step customer scenario may be expanded into the two-step scenario in Table 7.2 on the next page. State $0a$ is a newly expanded state as denoted by the alphabetic character tagged to the state number. In the right column, we show the enabling conditions of the steps, for example $8 > 2$ for state 0.

There are often multiple ways to expand a scenario. Table 7.3 on the following page is another expansion for the same customer scenario.

Step	***keyList***	**Condition**
0	$\langle 8, 4, 2, 6 \rangle$	$8 > 2$
0a	$\langle 2, 4, 8, 6 \rangle$	$8 > 6$
1	$\langle 2, 4, 6, 8 \rangle$	$2 \leq 4 \leq 6 \leq 8$

Tab. 7.2: **Z-scenario** *SortFourKeys* Expansion 1

Step	***keyList***	**Condition**
0	$\langle 8, 4, 2, 6 \rangle$	$8 > 4$
0a	$\langle 4, 8, 2, 6 \rangle$	$8 > 2$
0b	$\langle 4, 2, 8, 6 \rangle$	$8 > 6$
0c	$\langle 4, 2, 6, 8 \rangle$	$4 > 2$
1	$\langle 2, 4, 6, 8 \rangle$	$2 \leq 4 \leq 6 \leq 8$

Tab. 7.3: **Z-scenario** *SortFourKeys* Expansion 2

7.2.2 Duplicated Key Scenario

The customer also wants to sort lists with duplicated keys.

Step	***keyList***	**Condition**
0	$\langle 8, 4, 2, 6, 8, 4 \rangle$	
1	$\langle 2, 4, 4, 6, 8, 8 \rangle$	$2 \leq 4 \leq 4 \leq 6 \leq 8 \leq 8$

Tab. 7.4: **Z-scenario** *SortSixKeys*

Tables 7.5 and 7.6 on the next page show two possible expansions for the same customer scenario. Subscripts allow the specifier to track different instances of the same key value. The final state of the first expanded scenario has 8_b before 8_a while that of the second expanded scenario has 8_b after 8_a. Subscripts are not part of the original customer scenario. The relative order of the duplicated keys does not concern the customer.

7.2.3 Z

Our sorting scenarios have assumed that the keys are stored in sequences.

Step	***keyList***	**Condition**
0	$\langle 8_a, 4_c, 2, 6, 8_b, 4_d \rangle$	$8_a > 2$
0a	$\langle 2, 4_c, 8_a, 6, 8_b, 4_d \rangle$	$8_a > 4_d$
1	$\langle 2, 4_c, 4_d, 6, 8_b, 8_a \rangle$	$2 \leq 4_c \leq 4_d \leq 6 \leq 8_b \leq 8_a$

Tab. 7.5: **Z-scenario** *SortSixKeys* Expansion 1

Step	***keyList***	**Condition**
0	$\langle 8_a, 4_c, 2, 6, 8_b, 4_d \rangle$	$8_b > 4_d$
0a	$\langle 8_a, 4_c, 2, 6, 4_d, 8_b \rangle$	$6 > 4_d$
0b	$\langle 8_a, 4_c, 2, 4_d, 6, 8_b \rangle$	$4_c > 2$
0c	$\langle 8_a, 2, 4_c, 4_d, 6, 8_b \rangle$	$8_a > 2$
0d	$\langle 2, 8_a, 4_c, 4_d, 6, 8_b \rangle$	$8_a > 4_c$
0e	$\langle 2, 4_c, 8_a, 6, 8_b, 4_d \rangle$	$8_a > 6$
0f	$\langle 2, 4_c, 6, 8_a, 8_b, 4_d \rangle$	$8_b > 4_d$
0g	$\langle 2, 4_c, 6, 8_a, 4_d, 8_b \rangle$	$8_a > 4_d$
0h	$\langle 2, 4_c, 6, 4_d, 8_a, 8_b \rangle$	$6 > 4_d$
1	$\langle 2, 4_c, 4_d, 6, 8_a, 8_b \rangle$	$2 \leq 4_c \leq 4_d \leq 6 \leq 8_a \leq 8_b$

Tab. 7.6: **Z-scenario** *SortSixKeys* Expansion 2

SortingState

$keyList : \operatorname{seq} \mathbb{N}$

There is a common feature in all the expanded steps. The precondition is that two keys in the list are out of order. The postcondition is that the two keys are swapped. This common feature in the expanded steps can be generalised to schema *Swap* on Page 118. It has four local variables. Variables i and j hold indices and variables x and y hold the keys. The third and fourth predicates detect out-of-order keys in the list. The last predicate performs the swap.

Swap

$\Delta \mathit{SortingState}$

$i, j, x, y : \mathbb{N}$

$\mathit{keyList}(i) = x$

$\mathit{keyList}(j) = y$

$i < j$

$x > y$

$\mathit{keyList}' = \mathit{keyList} \oplus \{i \mapsto y, j \mapsto x\}$

Cooke discusses a measure decreased by a swap of out-of-order keys [39, page 322]. The measure eventually reaches zero when *Swap* cannot be invoked anymore. At that point, the negation of the 5-predicate conjunction is true. The list is sorted.

The out-of-order condition may be true for multiple pairs of keys in a state. The swap can apply to any one of the out-of-order pairs. In the above expansions, we do not insist that one pair of keys should have priority over another pair. The generalisation of the swapping steps leads to a schema that does not prescribe a particular order to perform the swaps. Therefore we consider the schema a specification.

7.3 Insertion Sort

In this section, we expand the customer scenario using insertion sort. We illustrate the sorting of five keys as follows [120]. The keys are positive integers. Value 0 at the beginning is not a real key. The sentinel value simplifies our job so there is no special handling to insert a key to the front. The first two keys, including sentinel value 0, are always sorted initially. The first iteration of the insertion sort algorithm begins with x_3.

Stage	x_1	x_2	x_3	x_4	x_5	x_6
3	0	$\downarrow 8$	$\boxed{7}$	2	4	6
4	0	$\downarrow 7$	8	$\boxed{2}$	4	6
5	0	2	$\downarrow 7$	8	$\boxed{4}$	6
6	0	2	4	$\downarrow 7$	8	$\boxed{6}$
	0	2	4	6	7	8

The boxed value is the key being moved and the downward arrow shows its new position. At stage j, the jth key is inserted in the correct position among the previously sorted $j - 1$ keys. After j stages, the first j keys are sorted. The action at each stage is very specific moving a specific key to a specific new place. There is only one route through the intermediate states from the initial state to the final state.

Step	$\boldsymbol{x}$	**Condition**
0	$\langle 8, 7, 2, 4, 6 \rangle$	
1	$\langle 2, 4, 6, 7, 8 \rangle$	$2 \leq 4 \leq 6 \leq 7 \leq 8$

Tab. 7.7: **Z-scenario** *SortFiveKeys*

Step	$\boldsymbol{x}$	$\boldsymbol{j}$	**Condition**
0	$\langle 8, 7, 2, 4, 6 \rangle$	1	$j = 1$
0a	$\langle 0, 8, 7, 2, 4, 6 \rangle$	3	$x_j = 7 \wedge 0 \leq 7 \leq 8$
0b	$\langle 0, 7, 8, 2, 4, 6 \rangle$	4	$x_j = 2 \wedge 0 \leq 2 \leq 7$
0c	$\langle 0, 2, 7, 8, 4, 6 \rangle$	5	$x_j = 4 \wedge 2 \leq 4 \leq 7$
0d	$\langle 0, 2, 4, 7, 8, 6 \rangle$	6	$x_j = 6 \wedge 4 \leq 6 \leq 7$
0e	$\langle 0, 2, 4, 6, 7, 8 \rangle$	7	$j > \mathit{length}(x)$
1	$\langle 2, 4, 6, 7, 8 \rangle$		$2 \leq 4 \leq 6 \leq 7 \leq 8$

Tab. 7.8: **Z-scenario** *SortFiveKeys* Expansion

The customer scenario in Table 7.7 is expanded to the developer scenario in Table 7.8 using insert sort. The first and last steps deal with the sentinel value 0. New variable j keeps track of the stage number.

Step	x	j	**Condition**	**Schema**
0	$\langle 8,7,2,4,6 \rangle$	1	$j = 1$	Initialise
0a	$\langle 0,8,7,2,4,6 \rangle$	3	$x_j = 7 \wedge x_1 \leq x_j \leq x_2$	Insert
0b	$\langle 0,7,8,2,4,6 \rangle$	4	$x_j = 2 \wedge x_1 \leq x_j \leq x_2$	Insert
0c	$\langle 0,2,7,8,4,6 \rangle$	5	$x_j = 4 \wedge x_2 \leq x_j \leq x_3$	Insert
0d	$\langle 0,2,4,7,8,6 \rangle$	6	$x_j = 6 \wedge x_3 \leq x_j \leq x_4$	Insert
0e	$\langle 0,2,4,6,7,8 \rangle$	7	$j > length(x)$	Finalise
1	$\langle 2,4,6,7,8 \rangle$		$x_1 \leq x_2 \leq x_3 \leq x_4 \leq x_5$	

Tab. 7.9: **Z-scenario** *SortFiveKeys* Generalised Expansion

Table 7.9 is similar to Table 7.8. The developer rewrites some values in the condition as indexed keys to facilitate the generalisation of conditions and actions to schemas. There is also a new column to show the name of the schema responsible for each expanded step.

The first two schemas *SortingState* and *InitSortingState* declare the state space and specify its initial state.

SortingState

$x : \operatorname{seq} \mathbb{N}$

$j : \mathbb{N}$

InitSortingState

$SortingState'$

$x' = \langle 8,7,2,4,6 \rangle$

$j' = 1$

Symbol $\frown$ stands for sequence concatenation. During the initialisation in schema *Initialise*, a value of zero which is less than the smallest keys is placed at the front of the key list.

Initialise

$\Delta SortingState$

$$
\begin{array}{l}
j = 1 \\
j' = 3 \\
x' = \langle 0 \rangle \frown x
\end{array}
$$

Now we come to schema *Insert* on this page. Symbol # is the size operator of sequences. Variable j holds the stage number. The first predicate ensures that j holds a valid key index. The second predicate requires the point of insertion i to be before j. The third predicate determines the value of i. The fourth predicate defines the new key list x' by inserting the jth key to the $(i+1)$th position. This is achieved by concatenating four components as follows. The first component $(1 \mathrel{..} i) \upharpoonleft x$ is a subsequence of original x from position 1 to i where symbol $\upharpoonleft$ stands for subsequence extraction. The second component $\langle x(j) \rangle$ is a 1-key sequence holding just x_j. The third component $((i+1) \mathrel{..} (j-1)) \upharpoonleft x$ is a subsequence of original x from positions $i+1$ to $j-1$. The last component $((j+1) \mathrel{..} (\#x)) \upharpoonleft x$ is a subsequence of x from position $j+1$ to the end.

Insert

$\Delta SortingState$

$i : \mathbb{N}$

$$
\begin{array}{l}
j \leq \#x \\
i \leq j \\
x(i) \leq x(j) \wedge x(j) \leq x(i+1) \\
x' = (1 \mathrel{..} i) \upharpoonleft x \quad \frown \quad \langle x(j) \rangle \quad \frown \quad ((i+1) \mathrel{..} (j-1)) \upharpoonleft x \quad \frown \\
\qquad\qquad ((j+1) \mathrel{..} (\#x)) \upharpoonleft x \\
j' = j + 1
\end{array}
$$

The Z function *tail* returns a sequence after the first item is removed. We use it to remove the phoney key 0.

Finalise

$\Delta SortingState$

$j > \#x$

$x' = tail\, x$

The expansion in this section allows no alternatives. More specific steps generalise to more specific schemas. The resulting schemas prescribe an order to perform the operations. Despite the use of Z notation, the schemas are really a program because of their embodiment of an algorithm. The sorting program terminates when no schema has *true* precondition.

7.3.1 More Scenarios

It is wise to consider more scenarios especially the ones with new conditions and actions. In stage 3 of the next scenario, x_3 with value 7 follows a smaller value 2. No move is required for the stage.

Stage	x_1	x_2	x_3	x_4
3	0	2	$\downarrow \boxed{7}$	3
4	0	2	$\downarrow 7$	$\boxed{3}$
	0	2	3	7

We need a schema to advance the stage number without moving the current key. Schema *Insert* is not satisfactory because it always moves the key to the left by at least one position.

Step	$\boldsymbol{x}$	$\boldsymbol{j}$	**Condition**	**Schema**
0	$\langle 2,7,3 \rangle$	1	$j = 1$	Initialise
0a	$\langle 0,2,7,3 \rangle$	3	$x_3 = 7 \wedge x_2 \leq x_3$	Advance
0b	$\langle 0,2,7,3 \rangle$	4	$x_4 = 3 \wedge x_2 \leq x_4 \leq x_3$	Insert
0c	$\langle 0,2,3,7 \rangle$	5	$j > \mathit{length}(x)$	Finalise
1	$\langle 2,3,7 \rangle$		$2 \leq 3 \leq 7$	

Tab. 7.10: **Z-scenario** *SortThreeKeys* Generalised Expansion

We write new schema *Advance* to fit the bill. Its first predicate checks that x_j is already in the correct position in the subsequence up to the jth position.

Advance

$\Delta SortingState$

$x(j-1) \leq x(j)$

$x' = x$

$j' = j + 1$

Sequences are the chosen data structure. Subsequence extraction and concatenation are the basic operations used. With insertion sort as the algorithm, the Z schemas describe a program.

7.4 Merge Sort

In this section, we expand a scenario using merge sort. Bags of sequences are the data structure. The operations used are "sequence to bag" conversion, bag union and bag difference. The customer scenario *SortFourKeys* in Table 7.1 on page 115 expands to the developer scenario in Table 7.11 on the following page with the help of new variable b holding a bag of sequences to be worked on. The first four steps in the expanded scenario create singleton sequences from the original key list. The enabling conditions of the steps at the first four states from 0 to $0c$ are a non-empty key list x. Due to the similarity in the conditions and actions, we would write schema *Split* to handle them. The next three steps, on states $0d$ to $0f$, merge sequences in b while maintaining the ascending order. Their enabling conditions are that b has at least two sequences. The finishing step on state $0g$ copies the only sequence in b to x which is the result expected by the customer. The condition of this step is that there is only one sequence in bag b and x is empty.

SortingState

$x : \mathrm{seq}\,\mathbb{N}$

$b : \mathrm{bag}(\mathrm{seq}\,\mathbb{N})$

Step	$\boldsymbol{x}$	$\boldsymbol{b}$	**Condition**	**Schema**
0	$\langle 8, 4, 2, 6 \rangle$		$\# x > 0$	Split
0a	$\langle 4, 2, 6 \rangle$	$\llbracket \langle 8 \rangle \rrbracket$	$\# x > 0$	Split
0b	$\langle 2, 6 \rangle$	$\llbracket \langle 8 \rangle, \langle 4 \rangle \rrbracket$	$\# x > 0$	Split
0c	$\langle 6 \rangle$	$\llbracket \langle 8 \rangle, \langle 4 \rangle, \langle 2 \rangle \rrbracket$	$\# x > 0$	Split
0d	$\langle \rangle$	$\llbracket \langle 8 \rangle, \langle 4 \rangle, \langle 2 \rangle, \langle 6 \rangle \rrbracket$	$(\llbracket \langle 8 \rangle \rrbracket \uplus \llbracket \langle 4 \rangle \rrbracket) \sqsubseteq b$	Merge
0e	$\langle \rangle$	$\llbracket \langle 4, 8 \rangle, \langle 2 \rangle, \langle 6 \rangle \rrbracket$	$(\llbracket \langle 2 \rangle \rrbracket \uplus \llbracket \langle 6 \rangle \rrbracket) \sqsubseteq b$	Merge
0f	$\langle \rangle$	$\llbracket \langle 4, 8 \rangle, \langle 2, 6 \rangle \rrbracket$	$(\llbracket \langle 4, 8 \rangle \rrbracket \uplus \llbracket \langle 2, 6 \rangle \rrbracket) \sqsubseteq b$	Merge
0g	$\langle \rangle$	$\llbracket \langle 2, 4, 6, 8 \rangle \rrbracket$	$(b \uminus \llbracket \langle 2, 4, 6, 8 \rangle \rrbracket) = \llbracket \rrbracket \land x = \langle \rangle$	Finish
1	$\langle 2, 4, 6, 8 \rangle$	$\llbracket \langle 2, 4, 6, 8 \rangle \rrbracket$	$x_1 \leq x_2 \leq x_3 \leq x_4$	

Tab. 7.11: **Z-scenario** *SortFourKeys* Generalised Expansion

InitSortingState

$SortingState'$

$x' = \langle 8, 4, 2, 6 \rangle$

$b' = \llbracket \rrbracket$

The steps are generalised to the following schemas. In schema *Split*, the first item of sequence x is removed and assigned to local variable k. A sequence, with just key k, is added to bag b.

Split

$\Delta SortingState$

$k : \mathbb{N}$

$\# x > 0$

$k = head\ x$

$x' = tail\ x$

$b' = b \uplus \llbracket \langle k \rangle \rrbracket$

In schema *Merge*, sequences p and q are in b. The third predicate ensures that sequence r has all keys from p and q. The fourth predicate ensures that keys in r are in ascending order. The last predicate removes old sequences p and q from b and adds new sequence r to the bag.

Merge

$\Delta SortingState$

$p, q, r : \operatorname{seq} \mathbb{N}$

$p \text{ in } b$

$q \text{ in } b$

$items\ r = items\ p \uplus items\ q$

$\forall i : 1 \mathrel{..} (\#r - 1) \bullet r(i) \leq r(i+1)$

$b' = (\ (b \uminus \llbracket p \rrbracket) \uminus \llbracket q \rrbracket) \uplus \llbracket r \rrbracket$

The first two predicates in schema *Finish* ensures that p is the only sequence in b. The third predicate copies the sequence to x.

Finish

$\Delta SortingState$

$p : \operatorname{seq} \mathbb{N}$

$p \text{ in } b$

$b \uminus \llbracket p \rrbracket = \llbracket\rrbracket$

$x' = p$

$b' = b$

7.4.1 More Scenarios

The sorting of a 1-key list has no practical significance. But we will test our program against it anyway.

Step	$\boldsymbol{x}$	$\boldsymbol{b}$	**Condition**	**Schema**
0	$\langle 6 \rangle$		$\#x > 0$	Split
0a	$\langle\rangle$	$\llbracket \langle 6 \rangle \rrbracket$	$(b \uminus \llbracket \langle 6 \rangle \rrbracket) = \llbracket\rrbracket \wedge x = \langle\rangle$	Finish
1	$\langle 6 \rangle$	$\llbracket \langle 6 \rangle \rrbracket$		

Tab. 7.12: **Z-scenario** *SortOneKey* Generalised Expansion

7.5 Specifications and Programs

Both specifications and programs formally describe computer behaviour. For most people, the main difference is that specifications describe the *what* and programs describe the *how*. This distinction is perspective-dependent. To the programmer of a financial application, mathematical models to simulate financial situations are the *what* and mathematical formulas to calculate various quantities in the models are the *how*. To the programmer of a C compiler, mathematical formulas in the input source programs are the *what*. The *what* to the compiler programmer is the *how* to the financial application programmer. The movable perspective makes the distinction between the *what* and the *how* a less than ideal tool to understand the difference between specifications and programs.

Hehner states that programs are implemented specifications [70] [71]. Programs are written in restrictive notations to facilitate execution. Hoare shares this differentiation of specifications and programs [79]. When logic is used for their representations, Kowalski considers efficiency to be the main difference between specifications and programs [96]. None of their views suggests that specifications and programs have a clear-cut distinction.

7.5.1 Three Decisions in Software Development

Our view is compatible with theirs. We treat specifications and programs as endpoints of a continuum. We place a set of schemas on the continuum based on three decisions that may be present in them. The decisions cover the data structures, operations and algorithms to be used. At the specification end, no decisions have been made. At the program end, all decisions have been made.

Figure 7.1 on the next page shows the positions of the four sorting descriptions on the continuum. Schema *SortSpec* uses sequences as the data structure. In other words, only one of the three major decisions have been made. Schema *SortSpec* is close to the specification end.

Schema *Swap* uses sequences for data structures and swaps for operations. Two of the three major decisions have been made. Without an algorithm of how out-of-order pairs are selected for swapping, it is still a

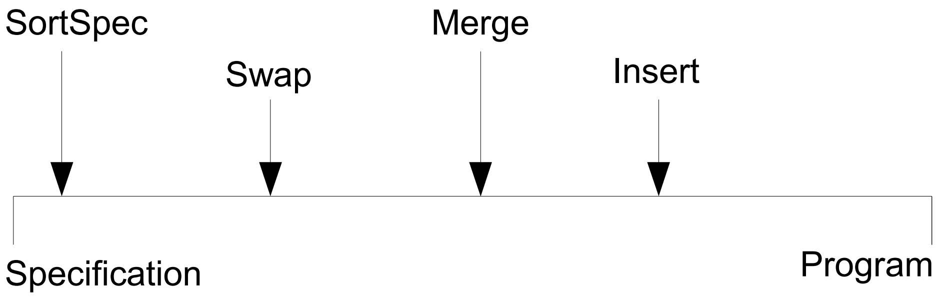


Fig. 7.1: The specification-program continuum

specification to most.

Schema *Insert* includes all three decisions on data structures, operations and algorithms. It is the closest to the program end among the four sorting descriptions.

Schema *Merge* uses bags of sequences, a number of bag operations and merge sort spanning the three major decisions. The fourth predicate of the schema ensures ascending order in the new sequence without describing how to build it from two smaller sequences. Decisions on algorithms are only partially made. Therefore we place schema *Merge* closer to the specification end than schema *Insert*.

7.6 Expansion

Customer scenarios have the states and variables that concern the customer. The corresponding developer scenarios have everything from the customer scenarios plus additional states and variables concerning only the developer.

Definition 7.6.1.

Let Q and Q' be sets of m customer and developer scenarios respectively such that $Q = \{q_1, q_2, \ldots, q_m\}$ and $Q' = \{q'_1, q'_2, \ldots, q'_m\}$.

$$
\begin{array}{l}
Q' \ \textit{expands} \ Q \\
\quad \textbf{\textit{iff}} \\
\forall\, i : 1 \mathinner{\ldotp\ldotp} m;\ t \in \operatorname{ran} q_i \bullet \exists\, x, y \in \mathbb{N} \bullet \\
\quad \langle t'_x, t'_{x+1}, \ldots, t'_y \rangle \ \text{in} \ q'_i \wedge \\
\quad \textit{X-TYPE} \lhd t \subseteq \textit{X-TYPE} \lhd t'_x \wedge \\
\quad \textit{Y-TYPE} \lhd t \subseteq \textit{Y-TYPE} \lhd t'_y \wedge \\
\quad \textit{I-TYPE} \lhd t = \textit{I-TYPE} \lhd (t'_x \cup t'_{x+1} \cup \ldots \cup t'_y) \wedge \\
\quad \textit{O-TYPE} \lhd t = \textit{O-TYPE} \lhd (t'_x \cup t'_{x+1} \cup \ldots \cup t'_y) \wedge \\
\quad \forall\, j, k \in x \mathinner{\ldotp\ldotp} y \bullet j \neq k \Rightarrow \\
\qquad \operatorname{dom} (\textit{I-TYPE} \lhd t_j) \cap \operatorname{dom} (\textit{I-TYPE} \lhd t_k) = \emptyset \wedge \\
\qquad \operatorname{dom} (\textit{O-TYPE} \lhd t_j) \cap \operatorname{dom} (\textit{O-TYPE} \lhd t_k) = \emptyset
\end{array}
$$

Step t in customer scenario q_i is expanded into a subsequence of steps $\langle t'_x, t'_{x+1}, \ldots, t'_y \rangle$ in the corresponding developer scenario q'_i. We use domain restriction symbol $\lhd$ extensively in the definition. The second conjunct requires the first expanded step t'_x in the subsequence to have a pre-state to match the pre-state of step t. The third conjunct requires the last expanded step t'_y in the subsequence to have a post-state to match the post-state of step t.

The fourth and fifth conjuncts require the expanded steps to cover the input and output of the original customer step. The last conjunct is a *for all* expression that prevents an input or output variable from being used more than once in two different steps in the expanded subsequence. This is necessary because an I/O variable is only used once in a customer step. The definition forbids behaviour altering expansions.

The number of developer scenarios must match the number of customer scenarios. The developer can selectively add details with extra scenario steps. The developer cannot remove steps or scenarios from the set of customer scenarios because it takes away behaviour requested by the customer. Not shown here, we can prove that a set of scenarios expands itself.

7.6.1 Combining Expansion and Observance

The definition of observance requires the schemas and scenarios to have the same granularity. A scenario step must be fulfilled by the invocation of one operation. Combining the notions of expansion and observance, we allow a customer scenario step to be effected by multiple operations.

A customer step may be too complex to be expressed in terms of our basic operations. Applying expansion, a customer step can be carved into smaller developer steps which are simple enough to be generalised in terms of our basic operations.

Customer scenario *SortFourKeys* has a lone step of a 3-way swap. It is relatively complex to express the enabling condition and action of a 3-way swap. The result of generalising such a 3-way swap into a schema would not be readily applicable to the sorting of other key lists. On the other hand, the condition and effect of a 2-way swap is simple. Appropriate repetitions of 2-way swap can sort various key lists. We decide to divide the sorting operation that involves a 3-way swap into multiple 2-way swaps.

Our expansions leading to the insertion sort and merge sort programs are more involved. The expansions are trivial to the author because he knows the algorithms. Without this prior knowledge, the discovery of a good algorithm to be used in the expansion requires insight and is likely a trial and error process. The use of developer scenarios does not replace creative thinking. It is a tool for documentation and separation of concerns. Creating software without developer scenarios is akin to doing mathematics without the use of a paper to write down the steps. It is less manageable and more error-prone. An expanded scenario allows the developer to document a computation in terms of actual data rather than variables. Unlike the customer scenario, the data used may be required by an implementation but does not concern the customer. The documentation of developer scenarios aids reasoning and communications.

An expanded scenario written during development does not represent a commitment. If the developer were unable to generalise an expansion, he or she would backtrack to try another expansion. If attempts at expansion and generalisation are to no avail, developers would go back to the customers for

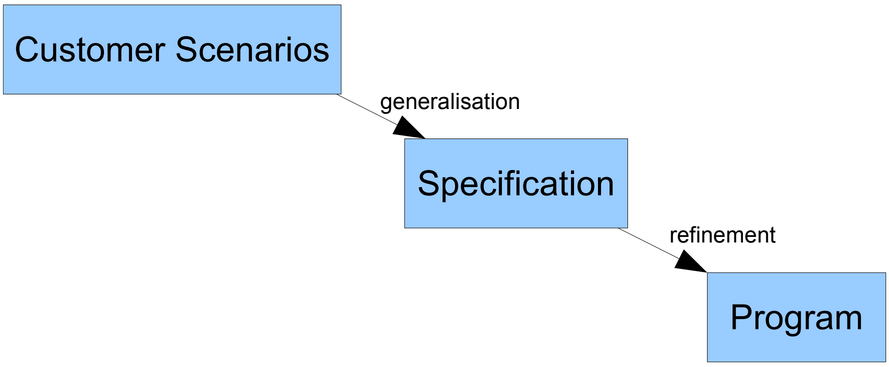


Fig. 7.2: Creating programs from specifications

a possible revision of the customer scenarios.

7.6.2 Complementing Formal Specifications

In previous chapters, we presumed that customer scenarios were used to create formal specifications which could be refined to programs. Refinement is a stepwise method to make a specification 'more deterministic' until it becomes an executable program [39, page 302] [10, page 20] as shown in Figure 7.2.

Scenario expansions provide an alternative approach to create programs from scenarios without going through formal specifications. Scenarios are expanded and then generalised to programs as shown in Figure 7.3 on the next page.

Implementation details of data structures, operations and algorithms are added to customer scenarios through expansions. The details are retained when we generalise developer scenarios to programs. We have introduced a new and rigorous approach to create programs without the use of formal specifications.

Should we skip formal specifications? One of their main contributions is the verification of programs. Throughout the thesis, we have been challenging the conventional wisdom of using formal specifications as the ultimate

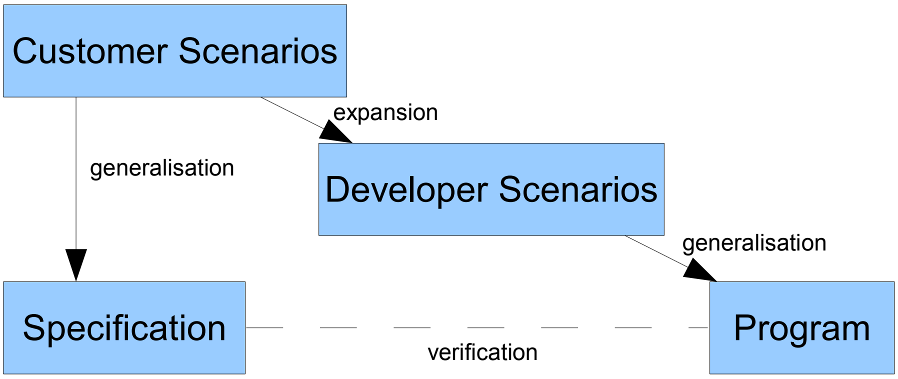


Fig. 7.3: Creating programs from developer scenarios

reference of software requirements. The creation of formal specifications lacks direct user involvement. On the other hand, the creation of concrete scenarios directly involves users. If generalisation is done correctly on a well-chosen set of scenarios, the resulting program can still be complete. Without a formal specification, we can verify that a program observes a set of customer scenarios in two steps. First, the program observes a set of developer scenarios. Second, the set of developer scenarios expands a set of customer scenarios. So far as correctness is concerned, we do not need a formal specification.

7.7 Chapter Conclusion

A description of computer behaviour contains three main major decisions: data structures, operations and algorithms. A description can be classified as a specification or a program based on the decisions it contains. We have developed Z operation schemas for sorting that stand on different positions on the continuum of specifications and programs. The more decisions we have included in a description, the closer it is to the end of programs on the continuum. However we are not always certain about the number of remaining decisions to be made. For example, a decision to use sets for the data structure may qualify a description as a program if we run it on a platform that directly supports sets. But if we have chosen a platform that only supports arrays, we need to decide how to implement sets on an array machine. There are more decisions to be made moving the description away from the end of programs.

Our approach supports the initial creation of customer scenarios which may be expanded to developer scenarios. Customer scenarios capture the what's; developer scenarios capture the how's. Concrete scenarios are a tool that allows customers and developers to clarify and document their thinking before their generalisation to specifications and programs respectively. Without a formal specification, we can still verify a program against the developer scenarios expanded from customer scenarios which capture our customer requirements.

Will scenario expansions result in overwhelmingly large number of steps? Not all customer steps need expansions. We expand just enough scenario steps to help us create the required operation schemas. The remaining unexpanded customer scenarios are still useful for requirements elicitation, documentation and testing. Developers are doing expansions in their brains anyway. Our approach documents developers' thoughts to make their mental process more tractable and to record them for others to see.

The chapter disseminates three development activities: writing customer scenarios, expanding them with implementation details and generalising the results into programs. The process separates the concerns of customers, developers and the generalisation activity. The results of the three activities are documented.

> A software development should be structured in some way - that there should be a separation of concerns. – M. Jackson [84, page 206]

8. NONDETERMINISTIC SCENARIOS

In last chapter, we explicate that the classification of software behaviour descriptions as specifications or programs is not exact. It depends on how many decisions on data structures, operations and algorithms have been made and remain to be made. The classification is influenced by the data structures and operations provided by the platform used. *Nondeterminism* describes the case in which a number of decisions are yet to be made. The use of this term on *generalised* software behaviour descriptions is seldom consensual. In this chapter, we use that term on concrete scenarios. Since concrete scenarios are *specific* rather than *generalised* descriptions, the meaning of nondeterminism may become easier to grasp.

A customer scenario is nondeterministic because algorithms have not been chosen yet. A fully expanded scenario is deterministic for it captures the developer's choice of algorithms. Nondeterministic scenarios are generalised to specifications while deterministic scenarios are generalised to programs. This chapter elaborates on nondeterminism with illustrations of a dice rolling simulator. An ordinary dice gives a random outcome from one to six. Our customer is a trickster who has a special requirement. He wants the dice to give random outcomes most of the time. Only occasionally, he would tamper with the outcome.

8.1 Deterministic Customer Scenarios

A set of scenarios is *deterministic* if and only if the same input parameters and pre-state always lead to the same output parameters or post-state.

Definition 8.1.1.

Let Q be a set of scenarios $\{q_1, \ldots, q_n\}$, Q is *deterministic*

iff

$$\forall\, i, j : 1 \mathinner{..} n \bullet \forall\, t_1 \in \operatorname{ran} q_i;\ t_2 \in \operatorname{ran} q_j \bullet$$
$$\mathit{X\text{-}TYPE} \lhd t_1 = \mathit{X\text{-}TYPE} \lhd t_2 \land \mathit{I\text{-}TYPE} \lhd t_1 = \mathit{I\text{-}TYPE} \lhd t_2$$
$$\Rightarrow \mathit{Y\text{-}TYPE} \lhd t_1 = \mathit{Y\text{-}TYPE} \lhd t_2 \land \mathit{O\text{-}TYPE} \lhd t_1 = \mathit{O\text{-}TYPE} \lhd t_2$$

A scenario is a sequence of steps. Another way to look at a scenario that it is a set of mappings from step numbers to steps. Steps are therefore the range of a scenario. Steps t_1 and t_2 map input, output, pre-state and post-state variables to values. They are two arbitrary steps in the scenarios. Symbol $\lhd$ stands for domain restriction. Expression $\mathit{X\text{-}TYPE} \lhd t_1$ only maps pre-state variables in step t_1 to values. We have similar expressions that restrict post-state, input and output variables. The implication requires any pair of steps to have the same output and post-state if their input and pre-state are identical.

Example

The user-controlled dice behaviour is exhaustively captured in the following set of six scenarios. Each scenario, enclosed in a pair of angle brackets, has a single step. There is no requirement for an internal state thus we do not have any state variables. The only variables in the steps are *trick*? and *outcome*! for input and output.

$$\begin{aligned}
\{\ &\langle\{\mathit{trick}? \mapsto 1, \mathit{outcome}! \mapsto 1\}\rangle,\\
&\langle\{\mathit{trick}? \mapsto 2, \mathit{outcome}! \mapsto 2\}\rangle,\\
&\langle\{\mathit{trick}? \mapsto 3, \mathit{outcome}! \mapsto 3\}\rangle,\\
&\langle\{\mathit{trick}? \mapsto 4, \mathit{outcome}! \mapsto 4\}\rangle,\\
&\langle\{\mathit{trick}? \mapsto 5, \mathit{outcome}! \mapsto 5\}\rangle,\\
&\langle\{\mathit{trick}? \mapsto 6, \mathit{outcome}! \mapsto 6\}\rangle\}
\end{aligned}$$

No two steps in the set of scenarios has the same input. Regardless of the choice of t_1 and t_2 in Definition 8.1.1 on the preceding page, the antecedent will always be *false*. The implication is trivially true and thus the set of scenarios is deterministic.

8.2 Nondeterministic Customer Scenarios

A set of scenarios is *nondeterministic* if and only if there exists two scenario steps in the set that have the same input and pre-state but different output or post-state. The output or post-state is controlled by factors not completely captured in the input and pre-state. The definition of nondeterminism is the negation of the definition of determinism.

Definition 8.2.1.

Let Q be a set of scenarios $\{q_1, \ldots, q_n\}$, Q is *nondeterministic*

iff

$$\exists\, i, j : 1 \mathrel{..} n \bullet \exists\, t_1 \in \operatorname{ran} q_i;\ t_2 \in \operatorname{ran} q_j \bullet$$
$$X\text{-}TYPE \lhd t_1 = X\text{-}TYPE \lhd t_2 \land I\text{-}TYPE \lhd t_1 = I\text{-}TYPE \lhd t_2 \land$$
$$(\ Y\text{-}TYPE \lhd t_1 \neq Y\text{-}TYPE \lhd t_2 \lor O\text{-}TYPE \lhd t_1 \neq O\text{-}TYPE \lhd t_2\ )$$

Example

The value of zero is not a valid outcome of rolling a dice. The user can use this value to indicate that a random outcome is desired. The finite nondeterministic behaviour can be specified with six other scenarios.

$$\begin{aligned}
\{\ & \langle\{trick? \mapsto 0, outcome! \mapsto 1\}\rangle, \\
& \langle\{trick? \mapsto 0, outcome! \mapsto 2\}\rangle, \\
& \langle\{trick? \mapsto 0, outcome! \mapsto 3\}\rangle, \\
& \langle\{trick? \mapsto 0, outcome! \mapsto 4\}\rangle, \\
& \langle\{trick? \mapsto 0, outcome! \mapsto 5\}\rangle, \\
& \langle\{trick? \mapsto 0, outcome! \mapsto 6\}\rangle\}
\end{aligned}$$

We combine this nondeterministic set of scenarios with the deterministic set of scenarios from the previous section to cover the complete behaviour of random and controlled outcome.

8.3 Nondeterministic Z Specifications

In Z, we define the types for input and output values as follows.

$$TRICK == \{trick : 0 \,..\, 6\}$$

$$OUTCOME == \{outcome : 1 \,..\, 6\}$$

The six scenarios for the deterministic behaviour are generalised to schema *TrickyRollSpec*.

$$\begin{array}{|l}
\hline
\textit{TrickyRollSpec} \\
trick? : TRICK \\
outcome! : OUTCOME \\
\hline
trick? \neq 0 \\
outcome! = trick? \\
\hline
\end{array}$$

The remaining six scenarios are generalised to schema *FairRollSpec*. Their nondeterministic behaviour is magically expressed with a disjunction of the six possible outcomes.

$$\begin{array}{|l}
\hline
\textit{FairRollSpec} \\
trick? : TRICK \\
outcome! : OUTCOME \\
\hline
trick? = 0 \\
outcome! = 1 \lor outcome! = 2 \lor outcome! = 3 \lor \\
\qquad outcome! = 4 \lor outcome! = 5 \lor outcome! = 6 \\
\hline
\end{array}$$

Our dice rolling simulator is specified by a disjunction of two schemas, one deterministic and one nondeterministic.

$$\textit{RollDiceSpec} \mathrel{\widehat=} \textit{TrickyRollSpec} \lor \textit{FairRollSpec}$$

8.4 Deterministic Programs

In the conventional use of formal methods, practitioners would obtain a program by refining the above specification. We use the approach introduced in the last chapter that creates a program from the expanded scenarios bypassing the formal specification. The program will be expressed in Z notation.

8.4.1 Developer Scenarios

We mimic the nondeterministic outcomes with a 6-member sequence, for example $\langle 4, 3, 2, 5, 1, 6 \rangle$. By cycling through the sequence, the user gets an impression of random outcomes especially when non-tricky rolls taken from the sequence are interspersed with tricky rolls. Following is our current set of twelve scenarios after expansion.

$$
\begin{array}{l}
\{\ \langle\{\mathit{trick}? \mapsto 1, \mathit{outcome}! \mapsto 1\}\rangle, \\
\quad\langle\{\mathit{trick}? \mapsto 2, \mathit{outcome}! \mapsto 2\}\rangle, \\
\quad\langle\{\mathit{trick}? \mapsto 3, \mathit{outcome}! \mapsto 3\}\rangle, \\
\quad\langle\{\mathit{trick}? \mapsto 4, \mathit{outcome}! \mapsto 4\}\rangle, \\
\quad\langle\{\mathit{trick}? \mapsto 5, \mathit{outcome}! \mapsto 5\}\rangle, \\
\quad\langle\{\mathit{trick}? \mapsto 6, \mathit{outcome}! \mapsto 6\}\rangle, \\
\quad\langle\{\mathit{trick}? \mapsto 0, \mathit{outcome}! \mapsto 1, \mathit{pos} \mapsto 5, \mathit{pos}' \mapsto 6\}\rangle, \\
\quad\langle\{\mathit{trick}? \mapsto 0, \mathit{outcome}! \mapsto 2, \mathit{pos} \mapsto 3, \mathit{pos}' \mapsto 4\}\rangle, \\
\quad\langle\{\mathit{trick}? \mapsto 0, \mathit{outcome}! \mapsto 3, \mathit{pos} \mapsto 5, \mathit{pos}' \mapsto 6\}\rangle, \\
\quad\langle\{\mathit{trick}? \mapsto 0, \mathit{outcome}! \mapsto 4, \mathit{pos} \mapsto 1, \mathit{pos}' \mapsto 2\}\rangle, \\
\quad\langle\{\mathit{trick}? \mapsto 0, \mathit{outcome}! \mapsto 5, \mathit{pos} \mapsto 4, \mathit{pos}' \mapsto 5\}\rangle, \\
\quad\langle\{\mathit{trick}? \mapsto 0, \mathit{outcome}! \mapsto 6, \mathit{pos} \mapsto 6, \mathit{pos}' \mapsto 1\}\rangle\}
\end{array}
$$

The first six scenarios need no expansion. They already contain enough information for generalisation to an implementation. The last six scenarios are expanded with a new variable *pos* which points to the outcome in the 6-member sequence $\langle 4, 3, 2, 5, 1, 6 \rangle$. If *outcome*! is 4 for a step, the value of *pos* should be 1 because that is the position of 4 in the sequence. The value of pos' would be $(\mathit{pos} + 1)$ except when we reach the end of the sequence, it will be reset to 1. The developer could have chosen a list with a different size and pattern. If we want to make the outcomes appear more random, a larger list would be used.

8.4.2 A Program in Z

The customer scenarios have been expanded with an algorithm to simulate the random behaviour by cycling through a list of outcomes. The result of generalisation will therefore be a program despite the fact that it is expressed in a formal specification notation Z. We define constant *OutcomeList* to hold the random outcome simulation sequence and type *POS* to hold valid positions in the sequence. The hash symbol # is the size operator for sequences.

$$OutcomeList == \langle 4, 3, 2, 5, 3, 5, 2, 4, 1, 6, 6, 1 \rangle$$

$$POS == \{pos : 1 \mathrel{..} \# OutcomeList\}$$

The types for input and output values are the same as those in the Z specification.

$$TRICK == \{trick : 0 \mathrel{..} 6\}$$

$$OUTCOME == \{outcome : 1 \mathrel{..} 6\}$$

Schema *Position* declares *pos* as a state variable.

$$\begin{array}{|l}
\multicolumn{1}{l}{\textit{Position}} \\ \hline
pos : POS \\ \hline
\end{array}$$

Schema *InitPosition* initialises *pos* to the beginning of the outcome list.

$$\begin{array}{|l}
\multicolumn{1}{l}{\textit{InitPosition}} \\ \hline
Position' \\ \hline
pos' = 1 \\ \hline
\end{array}$$

The user can use schema *SetPosition* to set *pos* to point to anywhere of the outcome list. Running this operation occasionally with varying input values increases the perception of randomness in the outcomes.

SetPosition

$\Delta Position$

$newPos? : POS$

$pos' = newPos?$

For fair rolls, the user indicates his or her intention of not controlling the outcome with an input of 0. The program returns an outcome from the list and updates *pos* to point to the next position of the list. If the end of the list is reached, *pos* is reset to the first position.

FairRollPgm

$\Delta Position$

$trick? : TRICK$

$outcome! : OUTCOME$

$trick? = 0$

$outcome! = OutcomeList(pos)$

$(pos < \#OutcomeList \land pos' = pos + 1) \lor (pos = \#OutcomeList \land pos' = 1)$

The program schema for the tricky rolls is the same as the specification schema.

TrickyRollPgm

$trick? : TRICK$

$outcome! : OUTCOME$

$trick? \neq 0$

$outcome! = trick?$

$RollDicePgm \mathrel{\widehat=} TrickyRollPgm \lor FairRollPgm$

8.5 Chapter Conclusion

The use of scenarios does not inhibit the expression of nondeterministic behaviour which can be captured by steps having the same input and pre-state but different output or post-state. Scenarios partition deterministic and nonderterministic behaviour. The two kinds of scenarios are generalised into separate schemas in a formal specification.

We show how scenarios are expanded to include details that concern only the developers. While the original scenarios document the thoughts of customers, the expanded scenarios document the thoughts of developers. The expanded scenarios facilitate communications much like the original scenarios. The difference is in the content being communicated.

Expansions make scenarios more deterministic. The generalisation of deterministic scenarios creates programs rather than specifications.

9. EMPIRICAL STUDY

The concrete scenarios presented earlier in this dissertation were written by the candidate alone. How will software pratitioners receive this approach of expressing requirements? More specifically, will people in the role of systems analysts be able to write concrete scenarios that are understandable by programmers for their creation of executable programs?

Section 1 of the chapter describes the objective of our empirical study. Section 2 discusses the choice of our subjects and application domains for the empirical study. Section 3 describes a few philosophical positions that empirical researchers can adopt. From the various positions, we have chosen the critical position. In the next few sections, we chronologically document and interpret the work of our three subjects. The chapter ends with a discussion of the lessons learned from the empirical study.

9.1 Objective of our Empirical Study

Our main objective in the empirical study is to find out if people trained in computing can write concrete scenarios. We do not directly test people's ability to read concrete scenarios. But if they can learn to write concrete scenarios, we argue that they must also be able to read concrete scenarios.

The subjects either know the PhD candidate before the empirical study or will get to know him personally in the course of the empirical study. They would not want to see the candidate to fail in the examination of his PhD research. Our subjects might be biased when asked how they feel about the concrete scenario approach. Therefore we refrain from asking their subjective feelings.

We are only interested in finding out if people can read and write concrete scenarios. We do not attempt a direct comparison of their readability and writability with alternative specification methods, at least not in this empircal study. We feel that the result of such a comparison will be heavily skewed by the training of the subjects and the maturity of the software tools supporting the approach. Due to the lack of a mature tool, we are not ready for a head-to-head comparision with competing methods.

Despite the limiting scope of our empirical study, we will be able to find out if people can use concrete scenarios as a means to communicate requirements. From the mistakes made by our subjects, we may learn how to fine-tune our notation and approach. Concrete scenarios may or may not be suitable for the entire requirements process. The findings in the empirical study may identify where we can put conccrete scenarios into good use.

9.2 Choice of Subjects and Application Domains

Finding appropriate subjects proves to be challenging. The subjects must afford the time to learn and use our approach. Most successful software practitioners are busy and expensive. As a compromise, one of our subjects is experienced but is semi-retired perhaps involuntarily due to his inability to keep up with the rapidly changing technology. Our two other subjects are fresh computing graduates with absolutely no real world experience. We introduce our three subjects as follows.

The first subject is Meng who has a Bachelor and a Master degree in computer science from the Georgia Institute of Technology. He has resided in Toronto after his graduation from Georgia Tech. With over a decade of programming experience in C and C++, Meng had also held the positions of analysts and project managers. He is in his early 50's. Though he is still very active in his social life, he has not been gainfully employed for about five years.

Our next two subjects are Kain and Lam who have recently completed the Bachelor in Computing at the Open University of Hong Kong (OUHK). OUHK was founded with government seed money but operate on incomes from tuition fees and endowments. OUHK was originally a distance learning institution but has ventured into face-to-face education in recent years. It has the lowest entrance requirements among all universities in Hong Kong.

The students were recommended by their program leader Dr. Andrew Lui as two of his better students. While looking for employment, they volunteer to take part in this research.

Application domains should be well understood by the subjects, the researcher and the readers. Thus we have chosen the browsing of an online catalogue, book borrowing at a library and checking out of a shopping cart.

9.3 Our Philosophical Position

Deductive reasoning and inductive reasoning are two paths to acquire new knowledge. Deductive reasoning employs logical proofs and deductions that have limited use in determining the receptivenes of practitioners to a new requirements specification approach. For that, we need inductive reasoning which involves drawing inferences from experiences and empirical data.

Quantitative empirical methods have four main characteristics: control, operational definition, replication and hypothesis testing [32]. Researchers control specific variables to isolate the cause of an effect. An operational definition describes the steps to obtain quantitative measurements. It can eliminate confusion in meaning and communication. Observations must be repeatable. Hypotheses are tested systematically. However the plethora of forces within individual human being and in the environment does not always afford the precise control or measurement by the researcher.

Qualitative methods offer a viable alternative to acquire knowledge about human behaviour. They stress holistic analysis rather than working with a few discrete variables [122]. Qualitative research can be meaningfully conducted on a small number of subjects. By focusing our resources, we can study to the level of details that might otherwise be infeasible. The collected details can also be used to design subsequent relevant quantitative research [54].

The researchers that use qualitative research adopt different underlying philosophical positions [109]. *Positivists* assume that reality is objectively given and can be described by measurable properties independent of the observers and their instruments [145]. *Interpretive researchers* assume that one can only access reality through social constructions such as language, consciousness and shared meanings. Interpretive studies attempt to understand phenomena through the meanings that people assign to them [22]. Instead of the previous two positions, we adopt the position of *critical researchers* who assume that social reality is historically constituted. Although people can consciously act to change their social and economic circumstances, their ablity to do so is constrained by various forms of social, cultural and political domination [28]. In simpler words, the ability of a subject to use concrete scenarios is influenced by his or her prior training.

9.4 Meng's First Batch of Work

After studying the *Concrete Scenarios Writing Guide* v1.0 in Appendix A on page 179, Meng was asked to write concrete scenarios for any user task on the Web site of a take-out / delivery restaurant. He picked the payment handling task. His work is shown in Appendix B on page 186. He wrote two scenarios for the task. Scenario 1 showed the acceptance of a cash payment. Scenario 2 showed the acceptance of a payment by Visa and the rejection of a payment by American Express. He tried to show which credit cards are acceptable with scenario 2. The unwarranted items in his scenarios are listed below.

1. In the English sentence template, he included data type information for each field. This is unnecessary but relatively harmless. It is however an evidence of the baggage that people bring from their prior experience when using this new approach.

2. He did not include the required input of payment type in the scenarios. This is comparable to the use of an undeclared variable in a computer program.

3. The scenarios of payment handling task should have side-effects on the system state otherwise we have no way of telling which orders have been paid for. But Meng did not record any side-effect in his scenarios.

4. He combined two examples of credit card payments into one scenario.

For clarity and readability, a scenario should describe one example only.

5. The name of scenario 2 is wrong. It was an obvious oversight when he copied and pasted from the *Concrete Scenarios Writing Guide*.

In a video conference, Meng complained that he was not clear what to include in concrete scenarios. He also asked for more examples of concrete scenarios. His suggestion, complaint and the mistakes found in his first batch of work result in a significantly revised guide version 4.2.

9.5 Meng's Second Batch of Work

After studying the revised *Concrete Scenarios Writing Guide* version 4.2 in Appendix C on page 188, Meng was asked to write scenarios for tasks relating to online shopping. He chose to write scenarios for the tasks of browsing the online catalogue, adding an item to shopping cart, and confirming an order. His second batch of concrete scenarios in revised syntax can be found in Appendix D on page 207. Despite our effort to improve the guide and the syntax of concrete scenarios, Meng did not write better concrete scenarios than before. Familiar mistakes were made along with new ones. Readers in a hurry should feel free to skip the detailed account of his mistakes by jumping to the concluding paragraph of this section on the next page.

Scenario *browse-item.1* on page 208 is problematic. For the input field *main-items*, Meng listed a range of possible values while only one value should be specified for the particular scenario. In the task definition, the catalogue table has two columns. But the same table is shown inconsistently in the scenario with one column and again with two columns. The two equalities in the relation part are also incorrect. How can *main-items* in the input equals *sub-items* in the first relation? They correspond to different levels in the catalogue.

The task description of *add-item* on page 209 may look acceptable on the surface. It has an input field not used in any scenario which is relatively harmless just like the declaration of an unused variable in a program. A closer look reveals that no field on the *orders* table can be used to identify the owner of a shopping cart. To solve this problem, a new column is needed on the *orders* table. Let's turn to scenario *add-item.1* on page 210. Its first relation shows that the selected item in the input equals the selected item in the stock but the corresponding equality between the input and the order is omitted. Another subtle mistake in the second relation is that the relational operator '$<$' should have been '$\leq$'. Meng made a small though annoying mistake when he copied and pasted scenario *add-item.1* to create *add-item.2*. He attempted to use a different item for the second scenario. But he had forgotten to change one of three occurrences of the item from 'War Game' to 'War & Peace'. He should have changed all three occurrences.

Scenario *purchase-item.1* on page 212 has an if-statement in the relation part. Its condition of "input.pickup = no" is not applicable to the scenario because the actual input value of *pickup* is *yes*. The use of if-statements goes against the principle of concrete scenarios to express requirements as examples not rules. The last two equalities in the relation part describe that the *stock* table should be updated but the table is not part of the scenario's system state. Variable *input.quantity* used in the last equality does not exist because *quantity* is not an input parameter of this task.

Meng has defined two instances of *purchase-item.1* with different input values. Meng apparently has forgotten to change the scenario number and description when he copied and pasted. The mistake of using an if-statement happened again in the second instance of *purchase-item.1*. He omitted a scenario for the *purchase-item* task with the input *pickup* being 'no' and the input *address* being non-blank.

In conclusion, Meng did not write scenarios based on an adequate set of tables. For example, he did not have a column to identify the shopper in an order. Meng did not use his tables consistently (with the same number of columns throughout). The lack of a syntax checker is partly to blame but Meng had made more than his fair share of mistakes. The repeated use of if-statements proves the difficulty he has to overcome in order to embrace the essence of concrete scenarios.

9.6 First Meeting with Kain and Lam

The *XML-based Concrete Scenario Writing Guide* v5.0 in Appendix E on page 213 introduces a syntax checker for concrete scenarios expressed in XML syntax. The guide also introduces an XSLT displayer for easy viewing of the XML-based concrete scenarios. Kain and Lam were given the guide to study one week prior to our first meeting. In the early part of the meeting, the students were given the opportunity to ask questions about the approach. Lam had not completely grasped the main concept of concrete scenarios. Kain helped us to explain to his peer.

Kain started using our notebook computer to write the concrete use case and scenario of borrowing a book from a library. It was Kain's idea to work on this task. He used the Liquid XML Studio software for about half an hour to create the concrete use case and a successful concrete scenario in Appendix F on page 246. We were happy to see Kain's satisfactory performance in writing his first concrete scenario.

Lam worked at the other end of the table using a scrapbook. He worked on the same user task as Kain. Initially, Lam broke down the main task into two subtasks of verifying borrower and verifying book. He asked us if that was a good start. We reiterated to him that the job of writing concrete scenarios is to document requirements by examples. We were not writing programs with stepwise refinement. After that, Lam was able to complete his work independently on his scrapbook. He worked on the visual form of concrete scenarios directly without going through the XML form. His resulting concrete scenario is similar to Kain's.

After observing their independent creation of a simple concrete scenario at the meeting, we believed that they knew enough of the approach to create additional scenarios without assistance. Kain suggested to us some changes leading to a more consistent XML syntax. We revised the syntax checking XML schema and the visual displaying XSLT. The two students were then given different user tasks to work on.

9.7 Lam's Work

As a continuation of the book borrowing scenario written at the meeting, Lam defined a failed attempt by a library user to borrow a book, a successful scenario of book returning and a successful scenario of book reservation. His work shown in Appendix G on page 252 is not perfect because there should be more scenarios for some tasks. Take the book reserving task as an example. We should consider the issue of fairness where there are two reservation requests for the same book. The second reservation request on a book could be different from the first request on the same book. Since Lam is an unpaid volunteer, we had decided not to pursue him beyond the scenarios he had written.

Lam's first scenario on page 252 was a failed attempt made by a library user to borrow a book. It failed because the borrowing quota has been exceeded. The relationships are clear and complete.

The second scenario on page 253 was a successful attempt to return a book. While the system state change is correct, a row number in a relationship is wrong.

```
3. system.user.1.borrowing_quota = system.user.1.borrowing_quota + 1
```

It should be corrected as follows.

```
3. system.user.2.borrowing_quota = system.user.1.borrowing_quota + 1
```

The scenario stated that the book is unavailable before it is returned.

```
4. system.book.1.available = "F"
```

It failed to state that the the book is available after it is returned. The following relationship should be added.

```
5. system.book.2.available = "T"
```

Lam made similar mistakes of using incorrect row numbers and omitting some relationships in the scenario of book reservation on page 254. There are more serious mistakes in the expressions referring to table cells.

```
3. system.available = "F"
4. system.booking = "T"
```

They should be corrected as follows.

```
3. system.book.1.available = "F"
4. system.book.1.book = "T"
```

As a summary, Lam made two kinds of low-level mistakes. The first kind of mistakes was just mentioned above which is the incorrect specification of fields in relationships. A powerful tool can provide a GUI that allows analysts to build relationships by selecting fields rather than writing potentially wrong XML expressions. The second kind of mistakes is the trivial omission of relationships. A semi-intelligent tool can check if every field has been mentioned in at least one relationship. This checking cannot detect the omission of multi-way relationships. Suppose a field is related to two other fields in two relationships. The tool will not alarm you if you have only omitted one because so far as the tool is concerned the field is covered by a relationship.

We have been examining Lam's scenarios for internal inconsistency. But what about inconsistency that spans multiple scenarios? Are the book borrowing scenarios consistent with the book reservation scenarios? We can chain a series of scenarios together so that the end state of one scenario is the begin state of the next scenario. Each scenario will become a step in a multi-step scenario which you have seen in Chapter 4. However the guide provided to our subjects does not treat the topic of multi-step scenarios.

9.8 Kain's Work

After our first meeting, we requested Kain to write concrete scenarios for the shopping cart check out operation. Even we wrote “shopping cart” in the email, we actually meant “online shopping cart”. Kain took the request literally and created scenarios that concern physical shopping carts. Despite the miscommunication, Kain demonstrated his ability to author concrete scenarios. He wrote the successful and failed scenarios of pushing a shopping cart into the unloading zone at the cash register. When writing requirements in words, analysts may or may not remember to write down the constraint of at most one shopping cart can be in the unloading zone at any time. The failed scenario by Kain illustrates this constraint with an example. Kain also wrote the scenario of checking out an item after the shopping cart has been placed in the unloading zone. The last scenario written by Kain is about the task of pushing the shopping cart off the unloading zone. As far as I can tell, all his concrete scenarios were correctly written. The relationships of the data fields contained in them are complete. By representing the *checking out cart id* as a simple data field, he limited his scenarios to the modeling of a single cash register. If he had represented the *checking out cart id* in a table, his concrete scenarios can model the concurrent operation of multiple cash registers.

9.9 Lessons Learned from Our Subjects

9.9.1 Reality of Old-School Analysts

Meng had at least 20 years of experience as either a programmer or an analyst. In the empirical study, he did not write his concrete scenarios based on an adequate set of data tables. In the relation part, he omitted some important data fields. He used the *if* keyword to make general statements about data fields. In the style of concrete scenarios that we endorse, data fields should be constrained to actual values. As a programmer, he wrote programs with rules in the form of if-then-else or looping statements. As an analyst, he wrote requirements as business rules. The programs and specifications he created in his professional career normally contained rules rather than examples. His prior experience and training primed him to write generalized rules instead of specific examples. When he needs to express requirements, rules may come to mind more naturally than examples. We cannot reasonably expect that a few hours or days of exposure to concrete scenarios approach can easily alter his practice developed over two decades. Though a good tool and quality instruction may be helpful, it will still be a major challenge for analysts to switch to expressing requirements in terms of examples only. If we insist on the use of concrete scenarios alone to express functional requirements, old-school analysts would doom the effort to failure.

9.9.2 Granularity

In Lam's first attempt to write a scenario of *borrowing a book* during the meeting, he tried to divide it into two parts: *verifying borrower* and *verifying book*. Lam mistakenly wrote concrete scenarios as if he was writing procedural programs. After having been corrected by us, Lam did not commit the same "mistake" again. In retrospect, Lam's division of a scenario into two parts may not be wrong. A complex scenario can be viewed as a succession of two or more simpler scenarios. Lam might just prefer to work at a different level of granularity. A good requirements notation should allow its users to select the level of details to work with.

9.9.3 Lack of a Bird's Eye View

When we first read the shopping cart scenarios written by Kain, we could not understand what Kain was trying to express. The bewilderment stemmed from our expectation to see scenarios about "online" shopping carts which are significantly different from scenarios about "physical" shopping carts. If a bird's eye view were available, the confusion would be avoided. For example, a UML statechart can show the big picture of state transitions caused by various events. The details of the state change in each transition can be documented by a concrete scenario. Concrete scenarios may be used with another notation that is more apt to present the big picture.

9.9.4 Completeness

Meng and Lam omitted data relationships in concrete scenarios. All three subjects omitted scenarios for the task they are specifying. As far as our subjects are concerned, concrete scenarios have not achieved completeness for them. However it does not mean that concrete scenarios cannot be helpful. We will come back to the topic of completeness in the concluding chapter.

9.9.5 Power of a Suitable Tool

Meng and Lam did not adapt immediately to the example-driven style of thinking embodied in concrete scenarios. Our primitive tool ensures the well-formedness of XML-based concrete scenarios. It weeds out some meaningless scenarios. Lam apparently had benefited from the tool. He did not make absurd mistakes as Meng did. Illegal syntax, like if-statements, is prohibited. However our current tool leaves a lot to be desired. It lacks a GUI that allows analysts to create data tables for use in concrete scenarios. It does not have a click-and-select feature that enables analysts always refer to the correct fields in data relationships. The development of a truly capable tool should be high on the agenda of our ongoing concrete scenario research.

9.10 Chapter Conclusion

We have learned that concrete scenarios should not be the only notation used by analysts to express requirements. Concrete scenarios may not be technically the best notation for all levels of granularity. For instance, the UML statechart may offer a better bird's eye view of the requirements. Even if concrete scenarios can match an alternative notation on technical ground, they may be considered inferior by an analyst due to his or her prior training. Of course, there is room for a good tool to support the use of concrete scenarios.

It is a coincidence that the three subjects spread over degrees of mastery of concrete scenarios: bad, mediocre and good. Capacity to express requirements in concrete scenarios should vary from person to person. From the types of mistakes made by our subjects, a few abilities are helpful.

- attend to details
- switch to declarative thinking (from procedural thinking)
- design data tables

Appendix I on page 259 is a guide to help programmers to derive Java programs from concrete scenarios. The guide also shows the derivation of automated tests encoded in JUnit which is a Java-based unit testing framework. We had promised the subjects not to take up an excessive amount of their time. After the subjects had written the concrete scenarios, we felt that we had used up our self-imposed quota of their time. The programming guide was not given to the subjects because we would like to postpone further empirical study. Once an appropriate set of tools has been built, we can build a stronger case to convince people to invest time learning and using the concrete scenario specification approach.

The usefulness of concrete scenarios rests on analysts' ability to write them as well as programmers' ability to read them. Our student subjects can read and write concrete scenarios. Work experience is not necessary and may not even be helpful as suggested by the case of Meng. The anecdotal evidence suggests that undergraduate computing education is a good preparation for the learning of concrete scenarios. We have not shown the cost-effectiveness of concrete scenarios in comparison with other requirements specification approaches. This will be a topic for future empirical studies with appropriate tools.

10. CONCLUSIONS

In this last chapter of the thesis, we summarise the work of early chapters in Section 10.1. Concrete scenarios that we have invented are evaluated in Section 10.2 on the issues of completeness, usability, scalability, costs and etc. Section 10.3 describes three popular requirements specification approaches. Section 10.4 compares concrete scenarios with the three competing notations on a number of desirable attributes we would like to find in requirements specifications. In Section 10.5, we outline research opportunities ahead. Section 10.6 discusses briefly some potential impacts of concrete scenarios on software development processes. Section 10.7 concludes the chapter and the thesis.

10.1 Synopsis of Previous Chapters

The incomprehensibility of formal specifications by customers sparked our research. After a survey of existing work in Chapter 2, we embarked on our quest of a precise and comprehensible requirements specification approach. We resolved to use concrete scenarios to describe user tasks performed in selected situations. An E-scenario consists of a number of successive states described in English with concrete data. If the states are precise, E-scenarios expressed in them are readily convertible to state-based formalisms. Since we have used Z notation to describe the states, we call them Z-scenarios.

In Chapter 3, we wrote a formal warehouse system specification which is a set of operation schemas generalised from scenarios. States in concrete scenarios may be expressed with English sentences. Simple sentences are used because of their manageability. We restrained ourselves to a small number of sentence templates to facilitate translation to a formal notation. Our scenarios should be more accessible to customers than the corresponding Z schemas because only a small subset of Z notation is used. If necessary, customers can read the equivalent English translations. The friendliness of concrete scenarios over formal specifications should boost customer involvement. It improves the odds that formal specifications reflect user requirements faithfully. We argue that the same benefits are extensible to other problem domains.

In Chapter 4, we specified a basic telephone system. Multi-step scenarios were used to capture interactions between phone users. They allowed developers to work on smaller steps one at a time. A ranking of scenarios based on importance guided developers to approach the system in an orderly manner according to customer priorities. For example, making basic phone calls was more important than conferencing. Thus we worked on the former scenarios first.

In Chapter 5, we enhanced the telephone system with conferencing. We started by considering new scenarios with the feature. We demonstrated how a specification could be kept up-to-date by revising or adding Z schemas. The traceability of schemas to scenario steps helps us quickly identify the affected schemas from updated steps or vice versa. We have shown how to update schemas for a modified state space. The chapter demonstrated how formal specifications derived from concrete scenarios may be maintained.

Chapter 6 defines the basics of verification. The building blocks are types of pre-state, post-state, input and output variables. A step is defined as a function that maps variables to values. A scenario is a sequence of steps where the post-state of a step matches the pre-state of the next step. The key result of the chapter is an observance relation between a set of operation schemas and a set of scenarios. The relation asserts that there is an operation schema that may be refined to effect every scenario step.

Chapter 7 works on sorting. We showed a sorting specification in first order logic and in Z. By the generalisation of scenarios, we created another sorting specification, an insertion sort program and a merge sort program, all in Z notation. Customer scenarios are expanded to developer scenarios with the augmentation of decisions about data structures, basic operations to use and algorithms. A customer step is expanded into a sequence of smaller and simpler developer steps often with the help of additional state variables. If the resulting operation schemas are generalised from customer scenarios with few design decisions, we call the set of operation schemas a specification. If the schemas are generalised from developer scenarios embodying most design decisions, the set of schemas would be considered a program. The key concept of this chapter is expansions. Developers can benefit from working with scenarios containing implementation details.

Chapter 8 continues the discussion of scenario expansion on a dice rolling simulator. The focus is on determinism. We captured deterministic and nondeterministic behaviour in separate sets of scenarios. Nondeterministic steps are expanded to deterministic steps before they can be simulated by a deterministic program. Observance and determinism are combined to define the notion of implementation. When a set of schemas observes a set of scenarios, the schemas may effect the scenario steps. When a set of schemas implements a set of scenarios, the schemas are guaranteed to effect the scenario steps.

Chapter 9 documents the empirical study of three subjects attempting to express requirements with concrete scenarios. The Z-inspired notations used in earlier chapters are replaced with notations in line with commonly used programming langauges such as Java to accommodate our subjects who have not been trained in formal methods. We want to emphasize that the concrete scenario approach is not tied to a particular set of notations. The notations we use in the concrete scenario approach need to be precise and comprehensible with the ability to express state changes. All our subjects can understand the concrete scenarios they read. But they do not master the writing skills equally well. Kain wrote correct scenarios naturally. Lam learned to write after some initial stumbles. Meng did not learn even after a few attempts. We have also discovered two major weaknesses in concrete scenarios. First, they do not present the big picture of user requirements. Second, they do not relate to high-level business goals.

10.2 Evaluating Concrete Scenarios

We shall evaluate concrete scenarios on their correctness, unambiguity, completeness, prioritisation, verifiability, modifiability, traceability, usability, scalability and costs.

10.2.1 Correctness

A requirements specification is **correct** if it accurately reflects the customer's needs. The customer's biggest concern is not how the system behaves. It is how the business goals may be accomplished through the system. Concrete scenarios can accurately describe the system behaviour externally through the input and output, and internally through the side-effects. But concrete scenarios are not being related to high-level business goals. We want to know to what extent the low-level requirements specified in concrete scenarios contribute to the attainment of high-level business goals. Concrete scenarios alone are inadequate. We need to incorporate them into another methodogy that deals with business goals. $i*$ described in Section 2.14.2 seems to be a generic enough methodology that may be compatible with concrete scenarios. Further work is required to explore this possibility.

10.2.2 Unambiguity

A requirements specification is **unambiguous** if every requirement in it has only one interpretation. A concrete scenario unambiguously describes an example of an operation by its input, output and side-effects. Programs are derived in a manual process of generalizing the data relationships in concrete scenarios. A unique interpretation of concrete scenarios depends on the fact that all significant data relationships have been identified. In other words, ambiguous concrete scenarios omit important data relationships.

Distinct data relationships in two concrete scenarios will be handled by two branches of a program as shown in the *Program Writing Guide* in Appendix I. If data relationships are missing, two fundamentally different scenarios will be handled by the same branch. When this happens, the programmer may suspect that important data relationships are missing. The programmer can then work with the analyst to discover the missing data relationships. This way of detecting ambiguity relies on the programmer's judgment of which scenarios should be fundamentally different. Empirical study is in order to find out how well programmers can make this judgment to catch ambiguously written concrete scenarios. We would also like to learn how well programmers and analysts can work together to discover the missing data relationships.

10.2.3 Completeness

A requirements specification is **complete** if it thoroughly covers functional and non-functional requirements. This dissertation uses concrete scenarios to express functional requirements only. Non-functional requirements must be expressed in other notations. To completely specify an operation, we need to write the concrete scenarios for all representative cases. We can determine if the concrete scenarios are complete with respect to an operation when we use them to derive a formal specification or a program as explained below.

In Section 3.9, we defined a user operation as a disjunction of several Z schemas. If the disjunction evaluates to true, we know that the specification of the operation can handle all possible input values in any system state. We can conclude that both our concrete scenarios and formal specification are complete.

In Appendix I *Program Writing Guide*, every program branch handles at least one scenario. If there is a branch, say the else-clause of an if-statement, found unrelated to any scenario, we know that we are missing a concrete scenario. Though concrete scenarios afford incompleteness detection, we do not know how well programmers can detect incompleteness in practice. Empirical study will be necessary to confirm this possibility.

A better alternative is not to detect incomplete concrete scenarios but to write complete scenarios in the first place. In our empirical study, the subjects did not produce a complete set of representative concrete scenarios. One possible explanation is that concrete scenarios do not contribute to completeness when first written. Before we come to that conclusion, we can try to improve our approach or tools to see if analysts can produce more complete scenarios on their first attempts.

10.2.4 Consistency

A requirements specification is **internally consistent** if no subsets of its individual requirements are in conflict. When a conflict exists, no programs can simultaneously satisfy all requirements stated in the specification. The exact nature of XML-based concrete scenarios permits certain automated consistency checks. One consistency check can make sure that all concrete scenarios for the same operation have the same parameter list. Another check can make sure that the same table is used consistently in all scenarios with an identical set of columns. More interesting is the case in which two scenarios produce different outputs or ending states from identical input and initial state. The requirements may be inconsistent or nondeterministic. See Chapter 8 for a treatment of nondeterminism. After consulting with customers, analysts will decide whether specifications are inconsistent or nondeterministic. Analysts will make the necessary corrections if the specifications are inconsistent. In conclusion, the exactness of concrete scenarios supports parsing which can discover inconsistencies. The use of concrete scenarios facilitates the creation of consistent requirements specifications.

10.2.5 Ranking by Importance or Stability

It is a good practice to **rank** requirements by importance or stability to facilitate their implementation in an appropriate order. Most requirements specification notations are neutral in this regard. Concrete scenarios are no exception. They neither assist nor hinder ranking.

10.2.6 Verifiability

A requirements specification is **verifiable** if for every requirement, there exists a cost-effective manual or automated process to check its fulfilment by the program. Concrete scenarios describe user operations by their input, output, pre-states and post-states. As we have demonstrated in Appendix I, each concrete scenario can be verified in a unit test. Concrete scenarios are fully verifiable.

10.2.7 Modifiability

A requirements specification is **modifiable** if it accommodates requirement updates. In conventional requirements specifications, some measures are useful for modifiability. Requirements may be written and organised to minimize crosscutting. Redundancies are minimized and allowed only when their benefits outweight the costs, perhaps for the sake of readability. A table of contents, an index and cross-references may also be used.

In the same way, good organization makes concrete scenarios more modifiable. Concrete scenarios are naturally organized by the operations that they belong to. With a good tool, this 2-tier organization will be fine for a system with a few operations and a few dozen scenarios. Practical systems could have a few hundred or even thousand scenarios. It quickly becomes unwieldy. We can consider adding tiers above operations to give more structure to the mostly flat 2-tier organization. High-level business goals are good candidates for the top tier. Under the business goals, we can add a tier or two of what we may call mid-level requirements notations such as Message Sequence Charts (MSC) or Statecharts. After integration with other notations, there will be four to five tiers in total allowing analysts to express requirements at any level they desire. Before we find an effective combination of notations along that line, requirements specifications in concrete scenarios are weak in modifiability.

Another thing working against concrete scenarios in their modifiability is the amount of details they capture. Information about system states found in concrete scenarios is rare among requirements notations. When requirements change, we have more details to update in concrete scenarios than in other requirements notations. This is an inherent disadvantage of concrete scenarios. A compromise act is to write concrete scenarios only for selected operations, such as complex operations that demand more analysis before coding.

10.2.8 Traceability

The *Program Writing Guide* in Appendix I demonstrates the development of program and test suites from concrete scenarios. Some data relationships in the scenarios are pre-conditions. They are tested in the conditions in if-statements or while-loops. Other data relationships are post-conditions. They are effected by assignments and/or database updates found in branches of the if-statements or at appropriate places in or around the loops. This style of program development supports strong traceability from scenarios to branches of if-statements as seen in the program on page 261. It is not a coincidence that we can derive programs this way. Data relationships must be either pre-conditions or post-conditions. Otherwise the data relationship should not be included in a concrete scenario at all. Data relationships concerning only the initial system state are pre-conditions to be tested in conditions of if-statements or while-loop. Remaining data relationships must concern the ending system state. They are post-conditions to be effected by appropriately placed assignments or method calls. We know what to do with every data relationship. We can therefore assert that the style of program development is generally applicable to concrete scenarios.

The development of test suites from concrete scenarios supports strong traceability as well. On page 267, we see three tests corresponding to three concrete scenarios. It is for the tester's convenience that we have made all three scenarios share the same pre-state. As a result, the three corresponding tests share the same initialisation method *initialiseTable*( ). In each test, the operation is called with the input values matching those in the respective concrete scenario. In the second and third tests *test_password_update_2*( ) and *test_password_update_3*( ) where there are no side-effects, we can just check the return Boolean values. If the return value is not Boolean, the checking may be slightly more involved. In test *test_password_update_1*( ) where there are side-effects, we can make calls to test the system state more thoroughly after the operation. For example, the test includes calls to the logon operation to see which password is good and which is not. When the concrete scenario is updated, we know exactly which test is affected and where we can make changes to keep the test suite up-to-date.

We have strong traceability between scenarios, programs and unit tests to assist programmers and testers to deal with requirement updates. But a notable weakness in traceability is that we are not providing backward traceability to high-level business goals. This weakness will be addressed if we can successfully incorporate concrete scenarios into a goal-oriented requirements specification approach such as $i*$.

10.2.9 Usability

A requirements specification is **usable** if analysts have no trouble writing it to express the requirements and programmers and testers can respectively read them to create programs and tests. In Chapter 9, we have learned of a few usability obstacles that our subjects encountered. Analysts may find it more natural to express requirements by general rules in the form of if-statements rather than by concrete examples in terms of data values. We realize that there should be different notations available to analysts to express requirements at a suitable level of abstraction. Concrete scenarios are a very low-level requirements notation. It will be counter-productive to only permit requirements to be expressed at that level. We should aim to integrate concrete scenarios smoothly into existing requirements notations for analysts to choose.

10.2.10 Scalability

A requirements notation is **scalable** if it can be effectively used to fully express the requirements of very large and complex systems. There are two aspects of scalability: size and complexity. Let us discuss the issue of complexity first. A concrete scenario for a complex operations may have more input and output values than an average operation. The side-effects and the data relationships will involve more fields and data tables than usual. However a concrete scenario is an example describing only a single case with one set of values. The concrete scenario will always be simpler than its implementing program that deals with many cases on a range of values. If the resulting program is manageable, the simpler concrete scenarios can only be more manageable as far as complexity is concerned. The tried-and-true technique of functional decomposition can break down arbitrary complex operations into simpler operations. We therefore do not think complexity will be a cause of concern in the scalability of using concrete scenarios.

We are more concerned about the impact of size. A large system supports many operations. A complex operation may spawn many concrete scenarios directly. If we break the complex operation into simpler operations before writing scenarios for it, we will have an even larger number of scenarios. We can try to understand the scalability of concrete scenarios based on experiences on software testing. Professional testers are not strangers to the management of several thousands of test cases in one system. Concrete scenarios contain similar type and amount of information found in white box tests. Therefore we think concrete scenarios are scalable given mature enough tools.

10.2.11 Costs

To estimate the costs of using concrete scenarios, we can start by analysing the information included in concrete scenarios. The input and output of concrete scenarios are required for the creation of black box tests. The system states described in concrete scenarios are useful information in the creation of white box tests. Consider *test_password_update_1* ( ) on page 267. The test checks the post-operation system state as well as the output. It demonstrates the information in concrete scenario can be reused for black box and white box testing.

Unlike black box testing, white box testing is not universally used. It is fair to say that in the creation of concrete scenarios, the specification of input and output values takes very little effort. Analysts expend most energy describing systems states and data relationships. When the added costs of white box testing are not justified in a software development project, the effort of writing concrete scenarios may not pay off. On the other hand, if a development tool helps us to reuse the information in the creation and maintenance of test cases, the costs of gathering the information in concrete scenarios would be bearable.

Concrete scenarios incur costs to train analysts and programmers. The costs depend on trainees' backgrounds, quality of instructions and maturity of the tools used. We would like to postpone further investigation in training costs. We no longer hold the position that concrete scenarios should be used to specify all operations across-the-board. The cost-effectiveness of concrete scenarios may be limited to critical or obscure operations.

10.3 Competing Requirements Notations

We have selected three commonly used requirements specification notations for a systematic comparison with concrete scenarios in the next section. In this section, we will introduce the notations and discuss them in light of the abstraction levels of requirements supported.

10.3.1 Requirements Definitions

Requirements definitions may be the most common form of requirements specifications documents. They are written in natural languages. Taken from Figure 3-14 of [43, Page 134] unmodified, Figure 10.1 on the following page is a list of functional requirements for a music store chain planning to sell their products online. Functional requirements are features of the system corresponding to product-level requirements. They are grouped under the user goals of (1) search and browse, (2) purchase and (3) promote.

10.3.2 Sequence Diagrams

UML Sequence diagrams are widely used. A sample from Chapter 2 is reproduced in Figure 10.2 on page 166. It shows three objects inside a system interacting with each other by sending and receiving messages. One or more of the objects are often users or external actors. A message is denoted by an arrow to represent an internal operation or a user task. Messages may have optional parameters.

10.3.3 Use Case

Use cases are a relatively new technique of expressing requirements. It was first formulated by Ivar Jacobson in the mid-80's and became popular in the 90's [87]. We have described it briefly in Section 2.1.

As shown in the example of Figure 10.3 on page 167, a central part of a use case description is the flow of events. An event can be an actor performing an action, for example providing an input to the system. An

1. Search and Browse

   1.1 The system will allow customers to browse music choices by predefined categories.

   1.2 The system will allow customers to search music choices by title, artist, and genre.

   1.3 The system will allow customers to listen to a short music sample of a music selection.

   1.4 The system will enable the customer to add music selections to a "favorite" list.

2. Purchase

   2.1 The system will enable the customer to create a customer account (if desired) that will store customer data and payment information.

   2.2 The system will enable the customer to specify the music choice for download.

   2.3 The system will collect and verify payment information. Once payment is verified, the music selection download process will begin.

3. Promote

   3.1 The system keeps track of the customer's interests on the basis of samples selected for listening and uses this information to promote music selections during future visits to the Web site.

   3.2 Marketing department can create promotions and specials on the Web site.

   3.3 Based on customer's previous purchases, music choices can be targeted to the customer on future visits to the Web site.

   3.4 On the basis of customer interests, customers can be notified of special offers on CDs.

Fig. 10.1: Requirements of a music store chain going online

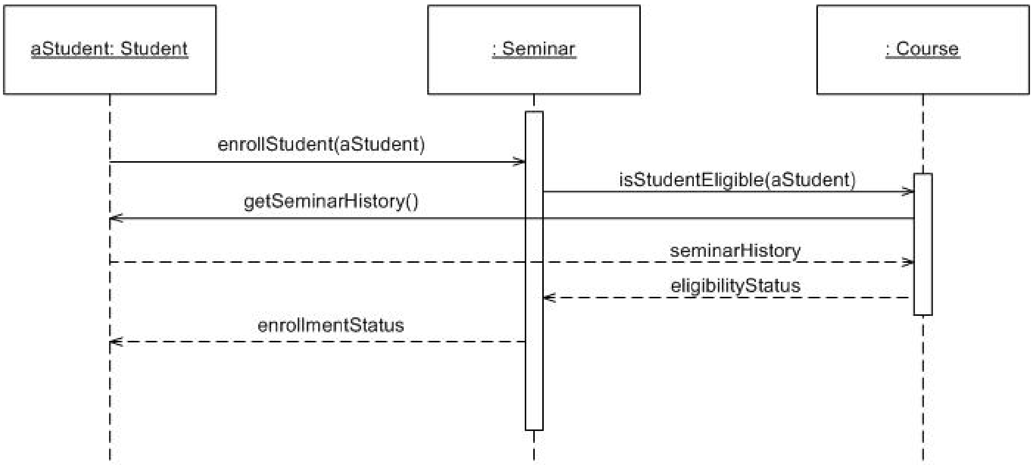


Fig. 10.2: A sequence diagram

event can also be a repsonse by the system, such as producing an output. Events are described in sentences to show subjects, actions, objects and parameters.

10.3.4 Requirements Levels Supported

Table 10.1 on page 168 summarizes how well each notation expresses requirements at a particular level. The five levels of abstraction are taken from [137] that was explained in Section 2.14.

It is a common practice for requirements definitions to have sections explaining the business goals and the linkage between them and user goals. But the use of natural languages does not ensure that business goals are well expressed and linkages are well established. The other three notations do not deal with business goals.

Requirements definitions group individual features under respective user goals. Sequence diagrams graphically relate messages in user tasks. Use cases list the events to achieve user goals. These three notations provide moderate support to domain-level requirements. Concrete scenarios do not provide a bird's eye view of how individual scenarios can be combined to attain user goals. Concrete scenarios cannot express requirements at the domain-level.

Use Case: Place Order
Precondition: A customer has logged on.
Flow of Events:

1. The customer selects Place Order.
2. The customer enters his or her name and address.
3. The customer enters product codes and quantities for items of his or her choice.
4. The system supplies a product description and price for each item.
5. The system keeps a running total of items ordered.
6. The customer enters credit card information.
7. The customer selects Submit.
8. The system verifies the information and saves a pending order.

Postcondition: The new pending order is saved on the system.

Fig. 10.3: A simple use case description

Features in requirements definitions are often described without clear and explicit representation of data input and data output. Requirements definitions only provide moderate support of product-level requirements. Sequence diagrams and use cases provide strong support of product-level requirements. Concrete scenarios cover product-level with input and output information for operations but they do not cover functional lists and use cases. Therefore concrete scenarios only support product-level requirements moderately.

Concrete scenarios provide stronger support of design-level requirements because of the description of system states and actual data used.

Table 10.1 on the following page shows a pattern of requirements definitions favouring the high-level, sequence diagrams and use case descriptions favouring the mid-level and concrete scenarios favouring the low-level.

10.4 Comparing Concrete Scenarios

In this section, we compare concrete scenarios with competing notations in their support of desirable characteristics for requirements specifications.

10.4.1 Correctness

Concrete scenarios tend to focus on low-level requirements. It is unclear how the low-level requirements actually contribute to the business goals.

Requirements Support	**Requirements Definitions**	**Sequence Diagrams**	**Use Cases**	**Concrete Scenarios**
Business-level	○			
Domain-level	○	○	○	
Product-level	○	●	●	○
Design-level		○	○	●
Code-level				

○ = moderate support, ● = strong support

Tab. 10.1: Requirements Levels Supported by Notations

When business goals are taken as the ultimate reference of requirements correctness, concrete scenarios are the weakest of all notations. The bird's eye view afforded by sequence diagrams and use cases helps the requirements to correctly reflect the business goals. Requirements definitions usually have sections in the documents to discuss business goals and how they are satisfied by individual requirements. Well-written requirements definitions tend to do best in describing the business goals and relating them to individual features and constraints.

10.4.2 Unambiguity

Requirements definitions are the weakest in their precision. Despite the fact that Figure 10.1 on page 165 is intended to be a textbook example, we can find many alternative ways to interpret the functional requirements. Take item 3.2 as an example. The requirement of "Marketing department can create promotions and specials on the Web site" can be satisfied by any existing Web site without doing any software development up front if static html files are used to hold the promotions. On the other hand, it can lead to a sophisticated database-driven application that can create promotions with many bells and whistles.

Sequence diagrams only show input and output of messages. Side-effects of messages on the internal state are not represented. This allows a message to be interpreted differently. Use cases descriptions suffer from the use of natural languages though their pre- and post-conditions can be helpful. The added details of system states in terms of data values make concrete scenarios the most precise notation of the four.

10.4.3 Completeness

It is easier to spot an omission in high-level requirements than in low-level requirements because of the better focus afforded by fewer items at the high-level. Sequence diagrams and use cases support requirements expressions at multiple levels of abstraction. Customers, analysts, programmers and testers have multiple opportunities to check for completeness.

Originally, we had expected concrete scenarios to give strong support on completeness. The completeness can be checked when scenarios are used to derive formal specifications or programs. We have not been able to support this with our empirical study. Based on the limited evidence that we have, we can only assign a low rating to concrete scenarios on completeness.

10.4.4 Consistency

The English used in the writing of requirements definitions allows conflicts to creep in unnoticed. You cannot easily tell how much two requirements have overlapped or how much they have contradicted.

The precision in sequence diagrams affords higher confidence in conflict detection. There is potential for automated consistency checking though only to the level of events not the level of detailed system states.

Use cases descriptions suffer in their precision due to the use of English. They make up some of the deficiency with their pre- and post-conditions. The partitioning of the functionality into use cases afford better manual consistency checking than do requirements definitions.

Detailed systems states and data values in concrete scenarios make them the best notation in ensuring a consistent requirements specification if we can assume the availability of appropriate tools to perform automated checking.

10.4.5 Verifiability

Similar to consistency, verifiability depends on a notation's precision. Expressed in a natural language, verifiability is not a strong suit of requirements definitions. Sequence diagrams state sequences of messages with senders and recipients which can be rigorously verified against an implementation. Use case descriptions have the pre- and post-conditions to compensate for the potentially imprecise event flows. Concrete scenarios are the most verifiable notation. As we have seen in the *Program Writing Guide* in Appendix I that data relationships in concrete scenarios can be turned into white-box tests which is a level of testing not required by many projects.

10.4.6 Modifiability

Requirements definitions and use case descriptions are both quite easy to modify though modifications may introduce inconsistency into the specification. Sequence diagrams are relatively easy to modify due to the small amount of information contained in them. Concrete scenarios are the worst in modifiability due to the sheer number of scenarios and the large amount of details inside each.

10.4.7 Traceability

All four notations support traceability given appropriate software processes and tools are used.

10.4.8 Usability

Usability of a requirements notation consists of two parts: writability and readability. The popularity of requirements definitions provides convincing argument for their writability and readability. Sequence diagrams and use cases require moderate amount of training and practice to write well. But they are both easy to read almost without training.

Concrete scenarios are also readable with minimal training. Concrete scenarios demand a paradigm shift from the writer. Without appropriate tool support, people may not be able to make the paradigm shift. Due to the weak tool support we can currently muster, we have not proved nor fairly assessed their writability.

10.4.9 Scalability

Requirements definitions, sequence diagrams and use cases have been widely used in requirements specifications for large systems. Scalability is well proven for them. One thing working against concrete scenarios is the amount of details in each scenario. Each operation spawns at least a few scenarios. A complex operation will have more scenarios and each with numerous data relationships. We are somewhat pessimistic about the scalability of concrete scenarios even before we field-test them on large systems. Modularization may help us deal with the scalability issue if we can successfully integrate concrete scenarios with mid- to high-level notations which we invent or select from existing notations.

10.4.10 Costs

Requirements definitions may be the best notation cost wise. They require little or no technical training for the readers and authors. The time and effort to read and write them are reasonable. They do not require expensive software. Use cases will cost slightly more in training and effort. Sequence diagrams will cost more in software licensing.

The costs of using concrete scenarios are higher in all areas. The main reason for the high costs is the added information in them. On a positive note, we think that there is a possibility for the costs of using concrete scenarios to be fully recouped in the saving of testing costs.

10.4.11 Comparison Summary

Comparisons of the four notations are summarized in Table 10.2 on the following page. The table shows that concrete scenarios stand out in the three areas of unambiguity, consistency and verifiability. Concrete scenarios match use case descriptions in two areas and trail in five. In their current form, concrete scenarios are not the best choice for general purpose requirements specification.

Support	**Requirements Definitions**	**Sequence Diagrams**	**Use Cases**	**Concrete Scenarios**
Correctness	●	○	○	◦
Unambiguity	◦	○	○	●
Completeness	○	○	○	◦
Consistency	◦	○	○	●
Verifiability	◦	○	○	●
Modifiability	○	○	○	◦
Traceability	○	○	○	○
Usability	●	○	○	◦
Scalability	●	●	●	○
Costs	●	○	○	◦

◦ = weak support, ○ = moderate support, ● = strong support

Tab. 10.2: Requirements Notations Comparison

10.5 Future Work

10.5.1 Integration with Mid-level Requirements Notations

Our empirical study shows that the lack of a bird's eye view is a problem with the use of concrete scenarios. The comparison with competing notations confirms this weakness of focusing solely on the low-level requirements.

We like the idea embodied in goal-oriented requirements approaches that low-level requirements should be shown to support business goals. Can we integrate concrete scenarios into a goal-oriented requirements specification notation such as $i*$? The five levels of requirements abstraction remind us that a direct integration may not be feasible. Goal-oriented approaches deal with the top two levels and concrete scenarios only support the third level moderately. We need additional work to bridge between a high-level goal-oriented requirements notation and low-level concrete scenarios.

The direct integration of concrete scenario with a mid-level requirements notation may give us a better payoff. Consider sequence diagrams which show bird's eye views of various related messages. Concrete scenarios may be written selectively for messages that require clarification. Sequence diagrams complement concrete scenarios and vice versa. Each notation does what it is best at. The integration may also solve the scalability issue of concrete scenarios. It may be too much work to specify the entire system with concrete scenarios. With the integration, a mid-level notation can specify the entire system with concrete scenarios saved for obscured parts.

In a similar way, concrete scenarios may be used to elaborate events in statecharts. This is also a good opportunity for concrete scenarios to go object oriented. States referred in concrete scenarios will be object states rather than system states. Object orientation also holds promise to the expression of concurrency in requirements.

After successful integration of concrete scenarios into other requirements notations, we could investigate how to best use concrete scenarios in the software development life cycle. For example, we can try to find out if concrete scenarios can facilitate parallel software development.

10.5.2 Tools Building

We have defined concrete scenario syntax in an XML schema. It ensures that important components are present in the concrete scenarios written by our subjects. The XML schema allows a typical XML tool on the market to check the well-formedness of concrete scenarios. But it does not prevent the writing of meaningless expression of "$x + 1 = 2$" even when $x = 0$. To address the shortcoming, we quickly put together a formatter in XLST to display the XML-based concrete scenarios neatly to facilitate visual checking by readers or writers.

Given more time, we should build a proper concrete scenario authoring program. Its users will write scenarios through a GUI which does not allow ill-formed scenarios to be written in the first place. This is better than detecting errors after they have been written. System states and object states must be used consistently in all scenarios. (Our subjects have used the same table but with different columns in different scenarios.) More intelligent checking can be built into the tool. For example, it should check that all data relationships are actually true in the scenario they belong. The objective of this tool is to make it easy to write scenarios and they must be well-formed and meaningful.

Another desirable tool would generate test cases from concrete scenarios. Concrete scenarios contain a superset of the information needed for black box testing. The test generation tool is effectively a sophisticated formatter that puts texts in the right places with appropriate texts inserted here and there. We are quite comfortable that such a tool would be feasible.

If we were to proceed with the research direction of integrating concrete scenarios with mid-level requirements notations, we would try to integrate our tools into other tools that specialize in mid-level requirements. This work however is not in our near-term agenda.

10.5.3 Further Empirical Studies

During this research, we have learned to appreciate the value of empirical studies. If we are serious about inventing or refining notations that people will find useful, we must not stop at the doable demonstration. We must go on to show the usability through appropriate empirical studies. Due to the primitive state of our method and tool, we are not ready for a quantitative empirical study now.

In [7], we have argued that concrete scenarios make the Z formal notation more accessible. The written feedback from the referees and the responses from the audience of the presentation seem to be in favour of our position. We could make a real contribution to learners of formal methods by putting together a package with tools and instructions to help them learn Z notations with concrete scenarios. Possibly the classroom setting will tolerate the primitive tool we can produce given our limited resources. Strides in this direction depend on an acceptable quality in the tools and instructions that we can provide and the successful recruitment of academic staff who are willing to use concrete scenarios in their formal method classes.

10.6 Potential Impacts on Software Development Processes

We are beginning to learn how concrete scenarios may be best used. Before our notation and tools mature, we may not fully appreciate the implications of using concrete scenarios. This section explores a few areas in software development where concrete scenarios may have an impact.

10.6.1 Users of Concrete Scenarios

Concrete scenarios will be written by people who normally have the title of **systems analysts** in North America. They may also be called *business analysts* or simply *analysts.* They are knowledgeable in the application domain. They fully appreciate the high-level business goals of the system being built. They know both the end-users of the system and the developers well and understand their languages. After all, they are the bridge between end-users and developers. They master the tools of requirements analysis. To write concrete scenarios, they must be able to switch paradigm to explain with concrete examples rather than generalized rules. When such an ideal person does not exist, two or more persons may split the task.

Concrete scenarios will be read by **programmers** to derive programs. They apply rigorous techniques to turn concrete scenarios into programs. The techniques have been demonstrated in the *Program Writing Guide* in Appendix I.

Testers also need to read concrete scenarios. Before test generation tools are available, testers must manually turn concrete scenarios into test cases. The techniques used by testers are more mechanical and easier than the techniques employed by programmers.

Assuming other notations are used to express mid-level and high-level requirements, system architects will not need to know the details in concrete scenarios.

Paying customers, end-user representatives and project leaders may not have the need to read concrete scenarios. It depends on how hands-on they are.

Concrete scenarios are used to enhance the precision of communications especially between analysts and programmers. If concrete scenarios serve their purpose, every stakeholder will be benefited.

10.6.2 Use of Formal Specifications

Advocates told us that "formal methods increase the cost of development" is a myth [67]. It was reported that the application of Z to IBM's CICS resulted in a 9 percent savings in development costs [25]. But few can dispute that formal methods may not be suitable or cost-effective for the development of all systems. If we can confirm from development projects that concrete scenarios improve the customers' understanding of formal specifications, we may widen the appeal of formal methods.

10.6.3 Test-Driven Development (TDD)

Dijkstra downplayed the significance of testing and regarded formal specifications as the ultimate reference of user requirements.

> Program testing can be used to show the presence of bugs, but never to show their absence. – E.W. Dijkstra [45, page 7]

Test-Driven Development (TDD) pratitioners write unit tests before the programs. They treat tests as a practical means to document requirements. Our position is compatible with theirs. But we do not want to be limited to black box tests of individual units. It is possible that concrete scenarios may become a variation of TDD.

10.6.4 Top-Down Versus Bottom-Up

According to conventional wisdom, we should approach computing problems systematically in a top-down manner. We thoroughly analyse requirements, design an architecture from them, break down the architecture repeatedly into small components until a component can be managed by an individual programmer.

The authority of the top-down approach is challenged by agile methods which include eXtreme Programing (XP), Test-Driven Development (TDD) and a lightweight project management method SCRUM. Agile methods do not attempt to learn all the requirements in details at the beginning of the project. They do not try to start with the most capable architecture for the problem at hand. They take a user story written in three sentences or less. With access to an onsite customer, they write the minimal program to implement the user story in two to three weeks. They refactor the code to rectify inappropriate program design decisions made earlier. Agile methods have a unmistakable bottom-up favour.

Some requirements analysis notations favour the top-down approach, for example, a data flow diagram (DFD) that breaks down a large process into a few smaller processes. Low-level concrete scenarios are neutral. They may become more important when developments are increasingly done in the bottom-up fashion.

10.7 Conclusion

We have argued that concrete scenarios can be a friendly alternative to formal specifications because they are more readable by people not trained in formalism. We have used concrete scenarios to specify a run-of-the-mill warehouse application, an interactive phone system, a nondeterministic dice and a computational sorting problem. We formally define the observance relationship of formal speciations by concrete scenarios. We conducted a qualitative experiment that involves subjects writing concrete scenarios. We learned that concrete scenarios are not as usable as we had expected. They do not portray the big picture of the system clearly. The vast amount of details in all concrete scenarios may be unmanageable for a large system. On the other hand, concrete scenarios offer better precision, consistency and verifiability than the competing notations. As they currently stand, concrete scenarios are useful for the clarification of potentially misunderstood system behaviour. With improved tool support and better integration of mid- to high-level requirements notations, concrete scenarios may become an excellent notation to specify all low-level functional requirements.

APPENDIX

A. CONCRETE SCENARIO WRITING GUIDE – V1.0

About this guide. This is the first tutorial written to help analysts to write concrete scenarios. It is a stripped-down version of the approach presented in earlier chapters. Data relationships which are normally a part of concrete scenarios are not required in order to ease the learning curve. The guide contains seven concrete scenarios for two user tasks: logon and id unlock. The tasks are applicable to an internal network or an email system. System states are first represented with English sentences. Interchangeable values in sentence templated are captured as columns on data tables. The guide uses a yellow background to highlight data that are important in the scenario. Values updated in a scenario are shown to the right of the old values with an arrow in between. Fields that appear before the table in the scenario are regarded implicitly as input. Fields appearing after the table are output. Analysts are expected to write concrete scenarios with an ordinary word processor without any assistance in the syntax.

A.1 Introduction

User requirements are often expressed in natural languages, for example English. Natural languages are ambiguous and subject to interpretations. Our research proposed to specify requirements precisely with concrete scenarios. In this tutorial, the approach is demonstrated with the creation of concrete scenarios for the user login functionality. There are four general steps as described in the following sections.

1. Use sentence templates to express system states

2. Use tables to capture the same information in the sentences

3. Name the scenarios for each user task

4. Describe each scenario based on its impact on the tables

A.2 Express system states in sentence templates

You will write a number of sample sentences that can capture the system state. Here are some examples.

1. User id francis is valid.
2. User id cudie is valid.
3. User id meng is locked.
4. User id francis has password hello246.
5. User id cudie has password 7there59.
6. User id meng has password family.
7. User id francis has accumulated 2 unsuccessful login(s).
8. User id cudie has accumulated 0 unsuccessful login(s).
9. User id meng has accumulated 4 unsuccessful login(s).

The sentences above only follow three templates. By varying the underlined values, we can derive an infinite number of sentences. With different combinations of sentences, we can capture different system states. This is not the only way to write the sentences. Another set of equivalent sentences, based on a single template, is shown below. It hardly matters whether you choose to use three simpler templates or a single slightly more complicated template. The key is that the sentences should be understandable.

1. User id francis is valid with password hello246 and 2 accumulated unsuccessful login(s).
2. User id cudie is valid with password 7there59 and 0 accumulated unsuccessful login(s).
3. User id meng is locked with password family and 4 accumulated unsuccessful login(s).

A.3 Extract variable parts of sentences into tables

While sentences can describe system states, they are too verbose. We prefer to use tables to capture equivalent information. The sentence template with four underlined fields gives rise to the following table of four columns. Meaningful column headings are used to enhance readability.

User Id	Status	Password	Accumulated failed login
francis	valid	hello246	2
cudie	valid	7there59	0
meng	locked	family	4

A.4 Identify successful and unsuccessful scenarios for a user task

We immediately think of a successful and a failed scenarios for the login task. It is helpful but not necessary to identify all the scenarios at the beginning.

A.5 Express scenarios with concrete data in table

A user task normally has input and output data. Our first scenario below shows a user trying to login with an incorrect password. He uses 'francis' as the id and 'goodbye' as the password. But from the table, we see that the correct password is 'hello246'. The data participating in the scenario are highlighted with a maron coloured font. A side-effect of the scenario is that the number of accumulated failed logins is incremented. The new value '3' after the scenario is shown to the right of the old value '2' separated by an arrow →. The output is a failure indicator.

Scenario id: 1
Scenario name: login failed for using an incorrect password
User id: francis **Password:** goodbye

User Id	Status	Password	Accumulated failed login
francis	valid	hello246	2 → 3
cudie	valid	7there59	0
meng	locked	family	4

Result: login failed

Starting from the same data table, if the correct password has been entered, we will have a successful login scenario. The number of accumulated login failure is reset to zero.

Scenario id: 2
Scenario name: login successful
User id: francis **Password:** hello246

User Id	Status	Password	Accumulated failed login
francis	valid	hello246	2 → 0
cudie	valid	7there59	0
meng	locked	family	4

Result: login succeeded

A.6 Adding more concrete scenarios to a user task

Using different data in the input or in the table in a scenario, we can discover new scenarios. It is especially easy to overlook the exceptional or erroneous scenarios, for example, trying to login with a locked id.

Scenario id: 3
Scenario name: login failed for using a locked id
User id: meng **Password:** family

User Id	Status	Password	Accumulated failed login
cudie	valid	7there59	0
meng	locked	family	4 → 5

Result: login failed

Another exceptional scenario is an attempt to login with a non-existent id. The paticipating data span a column instead of a row. There is no side-effect in this scenario.

Scenario id: 4

Scenario name: login failed for using a non-existent id

User id: gina **Password:** sister

User Id	Status	Password	Accumulated failed login
francis	valid	hello246	2
cudie	valid	7there59	0
meng	locked	family	4

Result: login failed

The above scenarios only result in the update of the accumulated number of login failures. We may also want to update the id status when the number of failures reaches four.

Scenario id: 5

Scenario name: login failed resulting in a locked id

User id: francis **Password:** candle

User Id	Status	Password	Accumulated failed login
francis	valid → locked	hello246	3 → 4
cudie	valid	7there59	0
meng	locked	family	4

Result: login failed

A.7 Adding more tables

As we continue to specify the complete system, we consider more user tasks. One of the tasks for the login functionality is to unlock previously locked id's. If the current system state is inadequate, we will need to repeat the earlier steps of writing sentence templates and creating tables before we can define scenarios for the new task. Unlocking is a privileged task that only managers should be able to do. We will create a new sentence template to capture the manager-employee relationship.

1. Manager cudie has employee(s) francis and meng.

2. Manager oliver has employee(s) jasper.

We shall need a new table as follows. We allow a list of user ids to be enclosed in a pair of square brackets [and]. Items inside a list are separated by commas.

Manager	**Employees**
cudie	[francis, meng]
oliver	[jasper]

To save space, we will only describe two scenarios for the unlock user task.

Scenario id: 6
Scenario name: unlock succeeded
Manager id: cudie **Employee id:** meng

User Id	**Status**	**Password**	**Accumulated failed login**
cudie	valid	7there59	0
meng	locked	family	4

Manager	**Employees**
cudie	[francis, meng]
oliver	[jasper]

Result: unlock succeeded

Scenario id: 7
Scenario name: unlock failed due to insufficient privilege
Manager id: oliver **Employee id:** meng

User Id	**Status**	**Password**	**Accumulated failed login**
oliver	valid	7there59	0
meng	locked	family	4

Manager	**Employees**
cudie	[francis, meng]
oliver	[jasper]

Result: unlock failed

A.8 Summary

Concrete scenarios capture usages of a system by examples. You first write sentence templates to capture system states in plain and understandable English. Then you capture the system state compactly in tables. Finally, you identify representative scenarios and describe them with actual input, output and the system states before and after. A complete system is specified incrementally, task by task, one scenario at a time. Each scenario represents a unique combination of input data and the system state. To write a complete specification, you need to exhaust all the representative combinations. As you build your requirements specification, you may need to revise your data to support new scenarios. What data to keep? What effect does a scenario has on the data? These are decisions to be made based on the computing requirements. Concrete scenarios are intended to be written by the systems analysts, precise enough for programmers to write programs and friendly enough for users to confirm if the requirements are accurately and completely captured.

B. MENG'S FIRST BATCH OF WORK

B.1 Express system states in sentence templates

Sentences that can capture the system state:

Order number (0001 to 9999) is valid with customer name (Alpha), address (Numeric & Alpha numeric), phone number (7 digits), food ordered (Alpha) and type of payment (Cash or Credit card)

B.2 Extract variable parts of sentences into tables

Order number	**Status**	**Customer name**	**Address**	**Phone number**	**Food ordered**	**Type of payment**
0001	valid	John Doe	123 Jane Street	555-6666	Soup	Cash
0002	valid	Jane White	456 John Street	666-5555 Chicken	Fried	VISA
0003	invalid		789 Center Street	888-9999	Hamburger	Cash

B.3 Identify successful and unsuccessful scenarios for a user task

Two scenarios for the Order task: successful (Valid) and unsuccessful (Invalid).

B.4 Express scenarios with concrete data in table

Payment is valid with cash or credit card

Credit card is valid with credit check

Scenario id: 1

Scenario name: Cash payment

Order number: 0006

Order number	**Status**	**Type of payment**
0006	valid	Cash

Result: succeeded

Scenario id: 2

Scenario name: Unlock succeeded

Order number: 0007 0008

Order number	**Status**	**Type of payment**	**Type of credit card**
0007	valid	Credit card	VISA
0008	invalid	Credit card	AMEX

Order number	**Credit check**
0007	Pass
0008	Fail

Result: succeeded & failed

C. CONCRETE SCENARIO WRITING GUIDE – V4.2

About this guide. This is the second tutorial provided to Meng to help him write concrete scenarios in the role of an analyst. The guide describes logon, password update and id creation tasks with twice as many concrete scenarios in the previous version. We assume that analysts will have no trouble working directly with data tables. The initial step of creating English sentences is therefore dropped though the sentences can be retrofitted at a later time if required by users. Task descriptions are written to serve as templates for scenario descriptions. Keywords **input** and **output** are used explicitly on the parameters. Hopefully, these changes could encourage analysts to give more thought to input/output parameters thus reducing the chance of omitting them in scenarios. Keywords **deleted** and **added** are used beside appropriate rows to identify outgoing and incoming data. They remind analysts to think about the side-effects in the concrete scenarios being written. Analysts are now required to include data relationships in the scenarios. Analysts will continue to write concrete scenarios with a word processor that offers no assistance in the use of correct syntax.

C.1 Introduction

This research proposed to express requirements with concrete scenarios. The goal of using concrete scenarios is twofold: readability and precision. In the proposed approach, a system is divided into a number of user tasks that users may want to perform. For example, browsing catalogue, adding items to a shopping cart and confirming an order are three common user tasks on online shopping Web sites. A user task may be performed in various ways giving rise to different outcomes. For example, ordering a book when it is out of stock is different from ordering the same book when it is in stock. We already have two scenarios for the same task of ordering a book.

In the existing literature, a scenario is an instance of a use case. Concrete scenarios adds to it the actual data used. An advantage of using concrete scenarios is that the requirements are so precisely expressed that they can be used as test data. In this tutorial, we will demonstrate concrete scenarios with the user logon functionality.

C.2 Logon Task

C.2.1 Task

Most user tasks have input and output data. To log on, the user inputs a user id and a password. The system responds with a success or failure result after comparing the input with the user information stored in an account table. There are three input and output fields: *user-id*, *password* and *result*. Each of them represents a single value. A table is distinguished from single-valued items by a box frame. The table is called *account* which has two fields: *user-id* and *password*. The logon task is described as follows.

Task:	logon
Description:	log on to the system
Input:	user-id, password
Output:	result
System:	account

user-id	password

C.2.2 Scenarios

A concrete scenario, described by actual values in the input, output and system data, is an example of performing a user task. The first concrete scenario description presented below has six components: scenario name, description, input parameters, output parameters, system data and relations between the input and system data. The name of a scenario is formed by the task name, followed by a dot '.' and a number, for example *logon.1*, *logon.2* and so on.

Scenario: logon.1

Description: the input matches a row in the system table

Input: user-id (francis), password (hello246)

Output: result (success)

System: account

user-id	**password**
cudie	7there59
francis	hello246
meng	family

Relation: input.user-id = account.2.user-id // input id matches
input.password = account.2.password // input password matches

Values of single-valued data items are enclosed in brackets. For example, the value of input *user-id* is ‘francis’. Scenario *logon.1* is a successful logon attempt as indicated by the result which is ‘success’. The acccount table is shown with three rows of data though only one row is actually involved in the two equalities of the relation component. The first equality says that the input *user-id* equals *user-id* in row 2 of the account table. The second equality says that the input *password* equals *password* in row 2 of the account table. We can place any comments preceded by // to explain the relations.

The second scenario for the task is a failed logon attempt in which the user inputs an incorrect password ‘goodbye’.

Scenario: logon.2

Description: the input password is incorrect

Input: user-id (francis), password (goodbye)

Output: result (failure)

System: account

user-id	**password**
francis	hello246
meng	family

Relation: input.user-id = account.1.user-id // input id matches
input.password $\neq$ account.1.password // but password does not

The third scenario for the task is also a failed logon attempt.

Scenario: logon.3

Description: the input user id does not exist

Input: user-id (jasper), password (alberta)

Output: result (failure)

System: account

user-id	**password**
cudie	7there59
francis	hello246
meng	family

Relation: // input user id does not match any user id on account table

input.user-id ≠ account.1.user-id

input.user-id ≠ account.2.user-id

input.user-id ≠ account.3.user-id

We have defined one successful scenario and two failed scenarios for the logon task. When a requirements specification is correctly written, it is common to see more failed scenarios than successful scenarios.

C.3 Password Update Task

C.3.1 Task

For the password update task, the input will be a user id, the old password and the new password. The output and system data are unchanged from the logon task.

Task: password-update

Description: change the password

Input: user-id, old-password, new-password

Output: result

System: account

user-id	password

C.3.2 Scenarios

The successful scenario of password update has a side-effect. The record with the old password is deleted. The record with the new password is added. To simplify our notation, we use a deletion followed by an addition to simulate a record update. The programmer is free to implement the scenario wth a direct update.

Scenario: password-update.1
Description: the input user id and old password match an existing user account
Input: user-id (meng), old-password (family), new-password (babygirl)
Output: result (success)
System: account

user-id	**password**	
meng	family	deleted
meng	babygirl	added

Relation: input.user-id = account.1.user-id
input.old-password = account.1.password
input.user-id = account.2.user-id
input.new-password = account.2.password

The two failed scenarios for the password update task resemble the failed scenarios for the logon task.

Scenario: password-update.2
Description: the input old password does not match the current password
Input: user-id (francis), old-password (goodbye), new-password (goodwill)
Output: result (failure)
System: account

user-id	**password**
cudie	7there59
francis	hello246

Relation: input.user-id = account.2.user-id
input.old-password $\neq$ account.2.password

Scenario: password-update.3

Description: the input user id does not exist

Input: user-id (jasper), old-password (calgary), new-password (edmonton)

Output: result (failure)

System: account

user-id	**password**
cudie	7there59
francis	hello246

Relation: input.user-id $\neq$ account.1.user-id

input.user-id $\neq$ account.2.user-id

C.4 Create Id Task

We have been using the account table to define scenarios for the logon and password update tasks. But where do the data in the account table come from? We need a user task to create user ids and passwords.

C.4.1 Task

Only privileged ids should be allowed to create other ids. We keep the privileged ids in a new table called *super-id*. Note that both tables are shown with initialization data: the user id 'admin' and an identical password. Once the system is configured, the systems operator is expected to change the obvious password to an obscure one. The third to fifth input parameters of the task are the information for the new id.

Task: create-id
Description: create a user id with an initial password
Input: creator-id, creator-password, id, password, privilege
Output: result
System: account

user-id	password
admin	admin

super-user

user-id
admin

This is not the only way to describe the system data for the task. An alternative is to add a *privilege* column to the account table as follows. An id can have either 'super' or 'normal' privilege. However we have chosen to use two tables instead for demonstration purpose. When necessary, the readers will know how to use multiple tables in one task.

account

user-id	password	privilege
admin	secret	super
cudie	hello	normal

C.4.2 Scenarios

The first scenario describes the creation of a super id. The creator id and password must match a row on the account table. The creator id must also be present on the super id table. The new id 'meng' is added to both tables.

Scenario: create-id.1

Description: create a super user id

Input: creator-id (admin), creator-password (secret)
id (meng), password (vacation), privilege (super)

Output: result (success)

System: account

user-id	**password**	
admin	secret	
meng	vacation	added

super-user

user-id	
admin	
meng	added

Relation: // matching creator's id and password
input.creator-id = account.1.user-id
input.creator-password = account.1.password
// creator is a super id
input.creator-id = super-user.1.user-id
// add the new id to account table
input.id = account.2.user-id
input.password = account.2.password
// add the new id as a privileged id
input.privilege = 'super'
input.id = super-user.2.user-id

Scenario *create-id.1a* is different from *create-id.1* in that no new row is added to the super id table. It is because the new id ‘cudie’ is not a privileged id as indicated by the normal privilege in the input.

Scenario: create-id.1a

Description: create a normal user id

Input: creator-id (meng), creator-password (vacation),
id (cudie), password (hello), privilege (normal)

Output: result (success)

System: account

user-id	**password**	
admin	secret	
meng	vacation	
cudie	hello	added

super-user

user-id
admin
meng

Relation: input.creator-id = account.2.user-id
input.creator-password = account.2.password
input.creator-id = super-user.2.user-id
input.id = account.3.user-id
input.password = account.3.password
input.privilege = ‘normal’

The next scenario shows that the attempt to create a new id from a normal id will fail. The reason is that the creator id ‘cudie’ is not on the super id table.

Scenario: create-id.2

Description: create a user id from a normal user id

Input: creator-id (cudie), creator-password (hello)
id (francis), password (francis), privilege (normal)

Output: result (failure)

System: account

user-id	**password**	
admin	secret	
cudie	hello	required
meng	vacation	

super-user

user-id
admin
meng

Relation: input.creator-id $\neq$ super-user.1.user-id
input.creator-id $\neq$ super-user.2.user-id

Another reason for the task to fail is that the id being created already exists.

Scenario: create-id.3

Description: create a user id with name conflicting with an existing id

Input: creator-id (admin), creator-password (secret)
id (cudie), password (pass1234), privilege (super)

Output: result (failure)

System: account

user-id	**password**
admin	secret
cudie	hello
meng	vacation

super-user

user-id
admin
meng

Relation: input.id = account.2.user-id

C.5 Password Expiry Feature

Software requirements evolve. The simple logon task defined earlier may be deemed insecure. We want the users to change their passwords periodically. We will remind users to change their passwords after one month of their previous password updates. If they ignore the reminders, two months after their previous password updates, ther ids will be locked. We are not defining a task. We are just modifying one or more previously defined tasks.

C.5.1 Revised Logon

The logon task needs to be revised to return the result of ‘please change your password’ when the password has been unchanged for over a month. However the tables used in the current system are inadequate. We need to store the date of the most recent password update for every user. This can be done by adding a column to the account table. The system also needs the current date available for comparison with the date of the most recent password update.

Task: logon
Description: log on to the system
Input: user-id, password
Output: result
System: current-date

account

user-id	password	updated-on

In the task description, we know that *current-date* is a single-valued data item because it is not followed by a box frame like the table *account*. The successful scenario is revised as follows.

Scenario: logon.1
Description: the input matches a row in the account table and the password has been updated within a month
Input: user-id (francis), password (hello246)
Output: result (success)
System: current-date (2009-Feb-03)

account

user-id	**password**	**updated-on**
francis	hello246	2009-Jan-25

Relation: input.user-id = account.1.user-id
input.password = account.1.password
current-date $\leq$ account.1.updated-on + 1 month

The new field of *current-date* and new column *updated-on* do not affect the failure of scenarios *logon.2* and *logon.3*. Other than the new field and column, nothing is changed. We do not produce their trivial revisions here.

According to the current date of the next scenario, Meng's password was last updated one month and a day ago. Therefore his logon attempt will succeed with the request to change password now.

Scenario: logon.4

Description: the password has been last updated for over a month
but not more than 2 months

Input: user-id (meng), password (family)

Output: result (success but change password now)

System: current-date (2009-Feb-03)

account

user-id	**password**	**updated-on**
cudie	7there59	2008-Nov-29
francis	hello246	2009-Jan-25
meng	family	2009-Jan-02

Relation: input.user-id = account.3.user-id

input.password = account.3.password

account.3.updated-on + 1 month < current-date

current-date ≤ account.3.updated-on + 2 months

In the next scenario, Cudie tried to logon but her password has not changed for over two months. Her logon attempt failed.

Scenario: logon.5

Description: the password has been last updated for over 2 months

Input: user-id (cudie), password (7there59)

Output: result (failure due to password not changed over two months)

System: current-date (2009-Feb-03)

account

user-id	**password**	**updated-on**
cudie	7there59	2008-Nov-29
francis	hello246	2009-Jan-25
meng	family	2009-Jan-02

Relation: input.user-id = account.1.user-id

input.password = account.1.password

current-date > account.1.updated-on + 2 months

C.5.2 Revised Password Update

When a user updates the password, the corresponding *updated-on* cell in the account table must be changed to the current date.

Scenario: password-update.1

Description: the input user id and old password match a user account

Input: user-id (meng), old-password (family), new-password (babygirl)

Output: result (success)

System: current-date (2009-Feb-03)

account

user-id	**password**	**updated-on**	
meng	family	2009-Jan-02	deleted
meng	babygirl	2009-Feb-03	added

Relation: input.user-id = account.1.user-id
input.old-password = account.1.password
input.user-id = account.2.user-id
input.new-password = account.2.password
current-date = account.2.updated-on

In the failed scenarios of password update, the value of *updated-on* is not changed. Thus the password expiry feature does not affect scenarios *password-update.2* and *password-update.3*. No revision to them is required.

C.5.3 Revised Create Id

When an id is created, its *updated-on* value in the account table should be set to the current date.

Scenario: create-id.1

Description: create a super user id

Input: creator-id (admin), creator-password (secret)
id (meng), password (vacation), privilege (super)

Output: result (success)

System: current-date (2009-Feb-03)

account

user-id	**password**	**updated-on**	
admin	secret	2009-Jan-28	
meng	vacation	2009-Feb-03	added

super-user

user-id	
admin	
meng	added

Relation: input.creator-id = account.1.user-id
input.creator-password = account.1.password
input.creator-id = super-user.1.user-id
input.id = account.2.user-id
input.password = account.2.password
current-date = account.2.updated-on
input.privilege = ‘super’
input.id = super-user.2.user-id

Scenario: create-id.1a

Description: create a normal user id

Input: creator-id (meng), creator-password (vacation),
id (cudie), password (hello), privilege (normal)

Output: result (success)

System: current-date (2009-Feb-10)

account

user-id	**password**	**updated-on**	
admin	secret	2009-Jan-28	
meng	vacation	2009-Feb-03	
cudie	hello	2009-Feb-10	added

super-user

user-id
admin
meng

Relation: input.creator-id = account.2.user-id
input.creator-password = account.2.password
input.creator-id = super-user.2.user-id
input.id = account.3.user-id
input.password = account.3.password
current-date = account.3.updated-on
input.privilege = ‘normal’

For a reason similar to the password update task, the failed scenarios of the id creation task do not need to be revised.

C.6 Guidelines of Writing Concrete Scenarios

Writing concrete scenarios is an incremental process. After the analyst has identified the user tasks to be supported by the system, he or she would write scenarios for each task. Step-by-step instructions follow.

C.6.1 Writing Task Descriptions

Task Give the task a short name which begins with an action verb. Multiple words should be concatenated with hyphens as in 'create-id'.

Description Give the task a description. Others should be able to tell two tasks apart by looking at their different descriptions.

Input List the input parameters the user needs to supply to the system to perform this task.

Output List the output parameters the user will get after performing the task.

System List the data items kept by the system that are required or affected by this task.

The input, output and system data may be single-valued or multi-valued. Tables are used to hold multi-valued data. To distinguish themselves from single-valued data items, multi-valued in tables are enclosed in framed boxes.

In our examples, the input and output have been single-valued data items. But it is also possible for them to have tables. For example, in a Web browsing task on an online bookstore, the output may be a table of book titles, author names, descriptions, prices and etc. In our examples, the system data have been mostly tables. But it is also possible to have single-valued system data like the *current-date* in the revised logon task.

C.6.2 Writing Scenario Descriptions

Varying combinations of input and system data give rise to different output. They can be captured in different concrete scenarios. For example, in the successful id creation scenarios, the privilege parameter may be 'super' or 'normal'. The value determines if the new id will be aded to the super user table. Therefore we need two scenarios for successful id creation.

Scenario The scenario name is made up of the task name, a dot '.' and a number, for example *create-id.1*. If two scenarios are sufficiently similar, you can emphasize their resemblance with the same number and a trailing letter, for example *create-id.1* and *create-id.1a*.

Description Others should be able to tell two scenarios apart by looking at their different descriptions.

Input Specify actual values for the input parameters in this scenario.

Output Specify actual values for the output parameters in this scenario.

System Specify actual values for the system data in this scenario.

Relation Specify the relationships between the input, output and system data.

The values of single-valued data items are enclosed in brackets following the data names. The values of multi-valued data items are listed in table format under column headings. Rows of data to be deleted from and added to a system table are denoted by keywords *deleted* and *added* respectively.

In the relation part, data are referred with the dot notation. Input and output data shall have the prefixes of *input* and *output* respectively before a dot. Data items without one of the two prefixes are assumed to be system data. Data from a table should be identified by a table name, followed by a row number, followed by a column name, all separated by a dot. You can relate two individual items by a relational operator. Line comments preceded with // may be used.

C.7 Summary

In this tutorial, we have written three user tasks relating to the system logon functionality. The first task was logon. You have learned that it is possible to have many scenarios for a single task. In particular, there may be more unsuccessful scenarios than the successful ones. The second task was password update. You have seen that it is possible to define new task using existing data tables. It is common for different tasks to share data tables in the system. The third task was id creation. We have used two tables to define one task. We also showed the initialization of a super id required for the task.

As requirements evolve, task and scenarios need revising. We might want to enforce periodic password update. The system should return reminders regarding password expiry. A few new scenarios were written to give those reminders with the help of the new column holding the recent password modification dates added to the account table. We also go through all the scenarios that use the modified account table for possible revisions.

D. MENG'S SECOND BATCH OF WORK

This is Meng's concrete scenarios documenting the tasks performed by a shopper on an online store. An evaluation of this work can be found in Section 9.5 on page 145.

D.1 Browse an online store

D.1.1 Task

Task: browse-item

Description: browse an online store

Input: main-items

Output: sub-items, detail-items

System: catalogue

main-items	sub-items

subCatalogue

sub-items	detail-items

D.1.2 Scenario

Scenario: browse-item.1

Description: browse an online store

Input: main-items (books, music, movies, games . . .)

Output: sub-items (art, biographies, computers, engineering . . .)

detail-items (digital photography, computer language, war games . . .)

System: catalogue

main-item
books
music
movies

⋮

main-items	**sub-items**
books	art
	biographies
	computers

⋮

subCatalogue

sub-items	**detail-items**
computers	digital photography
	computer language
	war games

⋮

Relation: // input item matches

input.main-items = subCatalogue.1.sub-items

// input item matches

input.sub-items = subCatalgoue.1.detail-items

D.2 Add an item to an online shopping cart

D.2.1 Task

Task: add-item

Description: adding an item to shopping cart at an online store

Input: selected-items, quantity, available

Output: result

System: orders

selected-item	quantity

stocks

selected-item	available

D.2.2 Scenario

Scenario: add-item.1

Description: add an item to the shopping cart

Input: selected-items (War Game), quantity (1)

Output: result (success)

System: orders

selected-item	quantity
War Game	1

added

stocks

selected-item	available
War Game	10

Relation: // input item matches

input.selected-item = stocks.1.selected-item

// quantity order is greater than available stock

input.quantity < stocks.1.avaliable

Scenario: add-item.2

Description: the input quantity required is out of stock

Input: selected-items (War & Peace), quantity (99)

Output: result (failure)

System: orders

selected-item	quantity
War & Peace	99

added

stocks

selected-item	available
War Game	3

Relation: // input item matches

input.selected-item = stocks.1.selected-item

// quantity order is greater than available stock

input.quantity > stocks.1.avaliable

D.3 Purchase item at an online store

D.3.1 Task

Task: purchase-item

Description: purchase an item at an online store

Input: email-id, pickup, address

Output: result

System: customers

email-id	pickup	address

D.4 Scenario

Scenario: purchase-item.1

Description: purchase an item in the shopping cart

Input: email-id (john@yahoo.com), pickup (yes), address ( )

Output: result (success)

System: customers

email-id	**pickup**	**address**	
john@yahoo.com	yes		added

Relation:

```
// save information
input.email-id = customer.1.email-id
input.pickup = customer.1.pickup
// save address for delivery
IF input.pickup = ‘no’
        input.address = customer.1.address
// Update stock
input.selected-item = stocks.1.selected-item
stocks.1.available = stocks.1.available − input.quanitity
```

Scenario: purchase-item.1

Description: purchase an item in the shopping cart

Input: email-id (jane@gmail.com), pickup (no), address ( )

Output: result (failure)

System: customers

email-id	**pickup**	**address**	
jane@gmail.com	no		added

Relation:

```
// save information
input.email-id = customer.1.email-id
input.pickup = customer.1.pickup
// save address for delivery
IF input.pickup = ‘no’ & input.address = ‘’
        put up error message
```

E. XML-BASED CONCRETE SCENARIO WRITING GUIDE – V5.0

About this guide. This is the third tutorial to help analysts to write concrete scenarios. Learning from Meng's not-so-successful experience, it becomes clear that the minimal tool we must provide is a syntax checker. It occurred to us that components in concrete scenarios can be expressed as customizable elements in XML documents. We expressed a document structure suitable for concrete scenarios in an XML schema stored in an XSD file. Analysts can use any XML editor available on the market to write concrete scenarios in XML. All industrial strength XML editors support the validation of XML documents against XSD files. We can now check the well-formedness of our XML-based concrete scenarios. We also created a stylesheet in Extensible Stylesheet Language Transformation (XSLT). It transforms the XML-based concrete scenarios to HTML displayable on Web browsers.

Analysts can use the first tool to check the syntax of XML-based concrete scenarios and use the second tool to display the XML-based scenarios in a more reader-friendly visual form. But our tools do not prevent the writing of some meaningless scenarios. Analysts can refer to a well-formed but non-existent field in a relationship. We have not built our tool with the ability to test the validity of relationships. We added an introduction to requirements specification for our inexperienced student subjects. They are reminded that relationships should cover every important field. In the visual display, outgoing data are now denoted with the brownish colour of soil and new data rows are shown in the green colour of sprout.

The previous guide shows how concrete scenarios are revised to cope with requirement changes. We have removed the discussion because it is unnecessary for this introductory guide.

E.1 The challenge of requirement specifications

During software development, requirements must be accurately communicated from the clients to the developers. Requirements are the foundation of most other work in a software project. A poor job in requirements specification will give rise to a solution not meeting the client's needs.

It is often necessary to capture requirements in writing. A use case is a user task. Use case descriptions can capture detailed functional requirements[1]. A use case may have a primary scenario and several alternative scenarios. Following is an example of the *log on* use case description.

Use case name:	Logon - primary scenario
Precondition:	System is up
Postcondition:	User is logged on
Event flow:	1. The system prompts the user to log on. 2. The user enters the correct user id and password. 3. The system verifies that the entered information.

Requirements expressed in English or any other natural language can be amibguious. Here is an example attributed to M. Jackson. He spotted two signs at the foot of an escalator in an airport.

1. Shoes must be worn

2. Dogs must be carried

The first sign required him to wear a pair of shoes before stepping on the escalator. By the same token, if he wanted to ride the escalator, he had to have a dog in his arms, right? Wrong! The second sentence really means that if he had a dog with him, he had to carry it while on the escalator.

[1] Software requirements are generally classified as two kinds: functional and non-functional. Functional requirements describe what the software can do, for example adding an item to a shopping cart on an online store. Non-functional requirements are other important qualities which may concern usability, reliability, performance, security and etc. "An average query must be completed in one second" is a non-functional requirement. Don't confuse use case descriptions with use case diagrams. Use case diagrams show the actors of use cases and interrelationships between use cases but no details beyond that.

But he was not required to have a dog. According to the first sentence however he was required to have a pair of shoes. The two sentences should be interpreted differently. If you want to improve the precision of the signs, you need to lengthen the sentences to deal with a number of situations. For instance, if you have 3 pairs of shoes with you, you are only required to wear one pair. On the other hand, if you have three dogs with you, you must carry them all. If your dogs are currently at home, you are not required to bring any of them to ride on the escalator. The two signs should not have caused many problems in practice. Since we know that not everyone has dogs, we will choose a more or less consistent context to interpret the simple sentences. Software requirements are far more complex. With clients and developers coming from different technical backgrounds, misunderstanding happens all the time.

E.2 Concrete use cases and scenarios

This research proposed to express detailed functional requirements with concrete scenarios. The goal is twofold: readability and precision. He describes a **concrete use case** with the kinds of input/output parameters and system data required for the user task. A **concrete scenario** is further described with the actual data used. A concrete scenario also shows how the actual output is related to the input and how the system data is modified by the scenario. A concrete scenario is not written as a generalized rule but an example. It is unclear how other people can manage this approach of specifying requirements. You are invited to study this tutorial that demonstrates the approach with a user logon application. You are then asked to apply the approach on another problem. Your experience with the approach will help us to determine the usability of the approach. Even if you were to have negative experience, your feedback is still welcome and may be helpful to refine our requirements specification techniques. Followings are the general steps in our concrete scenario specification approach. The first step produces a concrete use case. The second step produces a concrete scenario.

1. For each use case, determine

   (a) the kinds of system data involved

   (b) the kinds of input and output data involved

2. Identify representative examples of the use case, describe each example as a concrete scenario with

   (a) the actual input, output and system data used

   (b) the relationships between the data

Decisions on data design are made in concrete scenario specifications. It may be argued that concrete scenarios concern more than just requirements. The positioning of the approach as a requirement tool or a design tool however is not the focus of the current exercise. In the rest of this tutorial, we will be dealing with concrete use cases and concrete scenarios. When it is not likely to cause confusion, we may refer to them as use cases and scenarios for short.

E.3 Id creation

We will describe the scenarios for the use cases of *id creation*, *logon*, and *password update*. Given the incremental nature of the approach, it generally does not matter which use case do we start with. Let's start with the id creation task anyway.

E.3.1 Concrete use case

A concrete use case is described by five components: use case name, description, input parameters, output parameters and system state. The input of this use case consists of the creator's user id, password, new user id, its initial password and privilege. The output is a result indicating whether the task has been performed successfully. The system data consists of the current date and two tables. An individual data item is distinguished from a table by a pair of trailing brackets.

There is a system table called *account* to hold the user ids, passwords and the dates on which the passwords were last updated. The dates are useful if we want to remind users to change their passwords that have been unchanged long enough. There is another table called *super-id* that holds the ids with the privilege to create other ids. An alternative to this 2-table design is an additional column in the *account* table to hold the privilege of every id. Our decision to use the 2-table design is not due to its technical merits but our objective to demonstrate the use of multiple tables to represent system state.

A key activty in writing a concrete use case is to decide what system data are needed. We may make a decision on data organization that can adversely affect the writing of concrete scenarios. It may be necesary for us to reorganize the data or add missing data. These revisions could be tedious. Since we do not currently have good tools to support our approach, we will not show the revisions that may have taken place in order to keep our tutorial short.

Use case: create-id

Description: create a user id with an initial password

Input:

- creator-id ( )
- creator-password ( )
- id ( )
- password ( )
- privilege ( )

Output:

- result ( )

System:

- current-date ( )

account

user-id	**password**	**updated-on**

super-user

user-id

E.3.2 Concrete Scenarios

Concrete scenarios are really just examples of concrete use cases. The main difference between them is that concrete scenarios are described in terms of the actual data used. Values of individual data items are enclosed in brackets. Table data are listed under column headings row by row.

Given a system state and input parameters, a scenario specifies the output parameters and its side-effect to the system state. In scenario *create-id.1* below, the 'admin' user with password 'secret' attempts to create a privileged id for 'meng' with the initial password of 'vacation'. According to the system tables, the 'admin' id is defined on the system as a privileged id and its password matches the input password. Therefore we have a successful outcome. The scenario adds new rows to the system tables. New data rows are highlighted in a green background, for example row 2 for 'meng' is green in both tables.

Scenario: create-id.1
Description: create a super user id
Input:

- creator-id (admin)
- creator-password (secret)
- id (meng)
- password (vacation)
- privilege (super)

Output:

- result (success)

System:

- current-date (2009-Feb-03)

account

user-id	**password**	**updated-on**
admin	secret	2009-Jan-28
meng	vacation	2009-Feb-03

super-user

user-id
admin
meng

Relation:

1. input.creator-id = system.account.1.user-id
2. input.creator-password = system.account.1.password
3. input.creator-id = system.super-user.1.user-id
4. input.id = system.account.2.user-id
5. input.password = system.account.2.password
6. system.current-date = system.account.2.updated-on
7. input.privilege = ‘super’
8. input.id = system.super-user.2.user-id
9. output.result = ‘success’

Nine relationships are specified in this scenario. Similar relationships must hold between input, output and system data whenever privileged user ids are created. In this scenario, all relations happen to be equalities. Other relations are also possible, for example *less than*, *greater than* and etc. The syntax to express an individual item is a source name followed by an item name. The source can be *input*, *output* or *system*. A legitimate expression for an item in this scenario is *input.creator-id*. It refers to the value of an input parameter called *creator-id*. On the other hand, *output.id* is not allowed in this scenario because *id* is not defined in the output of the use case.

The syntax to express a cell in a table is slightly longer. It starts with a source name, followed by the table name, row number and column name, all separated by a dot. For example, the expression *system.account.1.user-id* stands for the *user-id* cell in row 1 on the system table called *account*.

Relations 1 and 2 specify that the input creator id and password match a user id and its password on the account table. Relation 3 specifies that the input creator id is a privileged id. Relations 4 and 5 specify the new user id and its password to match the corresponding input values. Relation 6 specifies that today's date is used initially as the last date that the password is updated. Relation 7 states that the new id is a privileged id according to the input. Relation 8 adds the new id to the privileged id table. Relation 9 specifies that the result is 'success'.

It is common for the relationships to cover all input parameters. If an input parameter is not mentioned in any relation, its value is not used by the scenario. The value of an output parameter is usually set to a constant or a value determined from the input and system data. It is theoretically possible for an output parameter not appearing in any relationship when we do not care about its value. Such a scenario is said to be *nondeterministic* which is uncommon in business applications.

In scenario *create-id.2*, the user of privileged id 'meng' adds a normal id 'cudie'. Scenarios *create-id.1* and *create-id.2* are very similar except for relation 8 in *create-id.1*. The new id is only added as row 3 to the account table but not added to the super-user table.

Scenario: create-id.2
Description: create a normal user id
Input:

- creator-id (meng)
- creator-password (vacation)
- id (cudie)
- password (hello)
- privilege (normal)

Output:

- result (success)

System:

- current-date (2009-Feb-10)

account

user-id	**password**	**updated-on**
admin	secret	2009-Jan-28
meng	vacation	2009-Feb-03
cudie	hello	2009-Feb-10

super-user

user-id
admin
meng

Relation:

1. input.creator-id = system.account.2.user-id

2. input.creator-password = system.account.2.password

3. input.creator-id = system.super-user.2.user-id

4. input.id = system.account.3.user-id

5. input.password = system.account.3.password

6. system.current-date = system.account.3.updated-on

7. input.privilege = ‘normal’

8. output.result = ‘success’

Scenario *create-id.3* specifies an unsuccessful attempt to create a new id from a normal id[2]. The failure is due to the creator id 'cudie' not on the super user table. No rows on the system tables are colored meaning that the scenario has no side-effects. The privilege of the new id in the input parameter is not mentioned in the relations. It means that the result will be the same regardless if the attempt is to create a super id or not.

Scenario: create-id.3
Description: create a user id from a normal (non-privileged) user id
Input:

- creator-id (cudie)
- creator-password (hello)
- id (francis)
- password (wong)
- privilege (super)

Output:

- result (failure)

System:

- current-date (2009-Feb-11)

account

user-id	**password**	**updated-on**
admin	secret	2009-Jan-28
meng	vacation	2009-Feb-03
cudie	hello	2009-Feb-10

super-user

[2] Modern programming languages, for example Java, have features to handle errors and exceptions reliably and elegantly. Software specifications should as well be written with enough attention to errors and exceptions. The use of concrete scenarios can be helpful in that regard.

user-id
admin
meng

Relation:

1. input.creator-id = system.account.3.user-id
2. input.creator-password = system.account.3.password
3. input.creator-id ≠ system.super-user.1.user-id
4. input.creator-id ≠ system.super-user.2.user-id
5. output.result = ‘failure’

Scenario *create-id.4* specifies an attempt to create a new id that fails for a different reason – the id already exists. This is possible under an imperfect administrative process, holders of the ‘meng’ id and the ‘admin’ id both try to create an id for the same new user. No table data are coloured. Therefore this scenario has no side-effects.

Scenario: create-id.4

Description: create a user id that already exists

Input:

- creator-id (admin)
- creator-password (secret)
- id (cudie)
- password (urgent)
- privilege (normal)

Output:

- result (failure)

System:

- current-date (2009-Feb-11)

account

user-id	**password**	**updated-on**
admin	secret	2009-Jan-28
meng	vacation	2009-Feb-03
cudie	hello	2009-Feb-10

super-user

user-id
admin
meng

Relation:

1. input.id = system.account.3.user-id
2. output.result = ‘failure’

E.4 Logon

E.4.1 Concrete use case

Our second use case is logon. It only needs two input parameters: user id and password. Only the *account* table is needed by this use case. We have an output parameter *message* used to remind users to change their passwords when they were unchanged for more than a month.

Use case: logon
Description: log on to the system
Input:

- user-id ( )
- password ( )

Output:

- result ( )
- message ( )

System:

- current-date ( )

account

user-id	**password**	**updated-on**

E.4.2 Concrete Scenarios

In scenario *logon.1*, user logs on with id ‘francis’ and password ‘hello246’. The password was last changed within a month as stated in relation 3. The logon scenario succeeds with a simple welcome message.

Scenario: logon.1
Description: log on to the system
Input:

- user-id (francis)

- password (hello246)

Output:

- result (success)
- message (welcome)

System:

- current-date (2009-Feb-03)

account

user-id	**password**	**updated-on**
cudie	7there59	2008-Nov-29
francis	hello246	2009-Jan-25
meng	family	2009-Jan-02

Relation:

1. input.user-id = system.account.2.user-id
2. input.password = system.account.2.password
3. system.current-date $\leq$ system.account.2.updated-on + 1 month
4. output.result = ‘success’
5. output.message = ‘welcome’

The second scenario is an unsuccessful logon attempt. The reason is captured in relation 2 where the input password ‘goodbye’ does not match the correct password ‘hello246’. There is no relation referring to the current system date because it does not matter in this scenario.

Scenario: logon.2
Description: log on with incorrect password
Input:

- user-id (francis)
- password (goodbye)

Output:

- result (failure)
- message (incorrect password)

System:

- current-date (2009-Feb-03)

account

user-id	**password**	**updated-on**
cudie	7there59	2008-Nov-29
francis	hello246	2009-Jan-25
meng	family	2009-Jan-02

Relation:

1. input.user-id = system.account.2.user-id
2. input.password ≠ system.account.2.password
3. output.result = ‘failure’
4. output.message = ‘incorrect password’

The third scenario of the use case is also a failed logon attempt. The first three relations show that the input id does not match any id currently defined on the system. We have no relations referring to the current system date or even the password because they are irrelevant in this scenario.

Scenario: logon.3
Description: log on with incorrect id
Input:

- user-id (jasper)
- password (alberta)

Output:

- result (failure)

- message (id does not exist)

System:

- current-date (2009-Feb-03)

account

user-id	**password**	**updated-on**
cudie	7there59	2008-Nov-29
francis	hello246	2009-Jan-25
meng	family	2009-Jan-02

Relation:

1. input.user-id ≠ system.account.1.user-id
2. input.user-id ≠ system.account.2.user-id
3. input.user-id ≠ system.account.3.user-id
4. output.result = ‘failure’
5. output.message = ‘id does not exist’

Scenario *logon.4* is a successful logon attempt. Unlike *logon.1*, the password has been updated for more than a month as stated in relation 3. Our client wants to remind the user to change the password.

Scenario: logon.4
Description: log on with password last updated for more than a month
Input:

- user-id (meng)
- password (family)

Output:

- result (success)
- message (please change password)

System:

- current-date (2009-Feb-03)

account

user-id	**password**	**updated-on**
cudie	7there59	2008-Nov-29
francis	hello246	2009-Jan-25
meng	family	2009-Jan-02

Relation:

1. input.user-id = system.account.3.user-id
2. input.password = system.account.3.password
3. system.account.3.updated-on + 1 month < system.current-date
4. system.current-date $\leq$ system.account.3.updated-on + 2 months
5. output.result = ‘success’
6. output.message = ‘please change password’

Relation 3 in scenario *logon.5* states that the password has not changed for over 2 months. Our client considers this to be unacceptable perhaps due to security reason. Note that relation 3 here is the negation of relation 4 in scenario *logon.4*. This is why scenario *logon.4* is a successful logon attempt but scenario *logon.5* is not.

Scenario: logon.5
Description: log on with password last updated more than 2 months
Input:

- user-id (cudie)
- password (7there59)

Output:

- result (failure)
- message (id locked because password unchanged over 2 months)

System:

- current-date (2009-Feb-03)

account

user-id	**password**	**updated-on**
cudie	7there59	2008-Nov-29
francis	hello246	2009-Jan-25
meng	family	2009-Jan-02

Relation:

1. input.user-id = system.account.1.user-id
2. input.password = system.account.1.password
3. system.current-date > system.account.1.updated-on + 2 months
4. output.result = ‘failure’
5. output.message = ‘id locked because password unchanged over 2 months’

E.5 Password Update

E.5.1 Concrete use case

The third and last use case we will write in this tutorial is password update.

Use case: password-update

Description: change the password

Input:

- user-id ( )
- old-password ( )
- new-password ( )

Output:

- result ( )

System:

- current-date ( )

account

user-id	**password**	**updated-on**

E.5.2 Concrete Scenarios

Scenario *password-update.1* replaces a row in the account table. The deleted row 2 with the old password is shown in the brown colour of soil. The new row 3 with the new password is shown in the green colour of sprout.

Scenario: password-update.1

Description: change the password

Input:

- user-id (meng)
- old-password (family)
- new-password (babygirl)

Output:

- result (success)

System:

- current-date (2009-Feb-03)

account

user-id	**password**	**updated-on**
francis	hello246	2009-Jan-25
meng	family	2009-Jan-02
meng	babygirl	2009-Feb-03

Relation:

1. input.user-id = system.account.2.user-id
2. input.old-password = system.account.2.password
3. input.user-id = system.account.3.user-id
4. input.old-password = system.account.3.password
5. system.current-date = system.account.3.updated-on
6. output.result = ‘success’

Scenario: password-update.2

Description: password update using incorrect current password

Input:

- user-id (francis)
- old-password (goodbye)
- new-password (goodwill)

Output:

- result (failure)

System:

- current-date (2009-Feb-03)

account

user-id	**password**	**updated-on**
francis	hello246	2009-Jan-25
meng	family	2009-Jan-02

Relation:

1. input.user-id = system.account.1.user-id
2. input.old-password $\neq$ system.account.1.password
3. output.result = ‘failure’

Scenario: password-update.3

Description: password update for a non-existing id

Input:

- user-id (jasper)
- old-password (calgary)
- new-password (edmonton)

Output:

- result (failure)

System:

- current-date (2009-Feb-03)

account

user-id	**password**	**updated-on**
francis	hello246	2009-Jan-25
meng	family	2009-Jan-02

Relation:

1. input.user-id $\neq$ system.account.1.user-id
2. input.user-id $\neq$ system.account.2.user-id
3. output.result = ‘failure’

E.6 Writing Guides

The previous sections demonstrated the use of concrete scenarios to express functional requirements precisely. Concrete scenarios are examples. Concrete scenarios are described by their exact effects on the system states. Relationships between data are expressed with relational operators, such as $=$, $<$, $\leq$ and etc.

In software projects, we have clients to provide software requirements at one end and programmers to translate the requirements into executable programs at the other. Systems analysts create requirements specifications which ideally should be understandable to the clients for their confirmation of correctness and completeness. The documents should also be precise for the programmers to do their job. System analysts are the bridge between clients and programmers. If concrete scenarios are used, systems analysts would be responsible for writing them.

E.6.1 Writing concrete use cases

A use case is a user task. We describe it in terms of a name, a description, input parameters, output parameters and the system data that it works on. XML is a language suitable for the expression of structured textual data. Followings are the XML fragment that we use to represent the *password-update* use case. Line numbers, not part of the XML document, are shown to facilitate explanation.

```
1  <use-case>
2    <name>password-update</name>
3    <desc>change the password</desc>
4    <input>
5      <item>user-id</item>
6      <item>old-password</item>
7      <item>new-password</item>
8    </input>
9    <output>
10     <item>result</item>
11   </output>
12   <system>
13     <item>current-date</item>
14     <table>
15       <name>account</name>
16       <column>user-id</column>
17       <column>password</column>
18       <column>updated-on</column>
19     </table>
20   </system>
21 </use-case>
```

Don't worry if XML is new to you. Its essence is quite simple. An XML document consists of a number of XML elements. An XML element is enclosed between matching begin and end tags. <use-case> on line 1 is a begin tag. </use-case>, differed with a slash character on line 21, is the matching end tag. XML elements may be related in a parent-child relationship. For example, the *use-case* element has five children *name*, *desc*, *input*, *output* and *system*. If we view the XML fragment as a tree, *name* and *desc* elements on lines 2 and 3 are leaf nodes. They contain data without any child elements beneath them. The *input* element on lines 4 to 8 is an intermediate node with three children of *item*. An *item* element, for example line 5, is a leaf node used to hold a single value. The *system* element on lines 12 to 20 has two children: an item and a table. The *table* element from lines 14 to 19 has four children: a name and three columns. Study the XML fragment side by side with the concrete use case of *password-update* to understand how the two correspond.

It is not enough for the system analyst to master our XML syntax. He or she has to identify the kinds of input and output parameters required for the use case. The analyst also has to choose a logical data structure, usually with some tables, to express the system state.

E.6.2 Writing concrete scenarios

Varying combinations of input and system data produce different output. Each combination is captured in a separate concrete scenario. For instance, by varying the value of input parameter *privilege* as 'super' or 'normal', we have two scenarios of successful id creation. Scenario *create-id.1* adds the new id to the super user table but scenario *create-id.2* does not.

Based the concrete use case's XML description, we can make the following changes to create a concrete scenario, for example scenario *password-update.1* in XML coming up shortly.

- Lines 1 and 68 – Change the enclosing *use-case* tags to *scenario* tags.
- Line 2 – Specify the id number of scenario with the *id* attribute. Note that In XML double quotes around attribute values are compulsory.
- Line 3 – Specify an appropriate description.
- Lines 4 to 8 – Specify the value of each input item with the *value* attribute.

- Lines 9 to 11 – Specify the value of each output item.

- Line 13 – Specify the value of each system item.

- Lines 19 to 33 – Specify each row on system tables. For each system table, at least for some scenarios, I encourage you to include non-participating data. Failure to do so may require you to revise the table design in the future.

- Lines 19 to 23 – We have three cells in a row to match the three columns on the table. Without the status attribute, this row of data is on the table before and after the scenario.

- Line 24 – The row status with value “old” states that the row exists before the scenario but deleted afterward due to its side-effect.

- Line 29 – The row status with value “new” states that the row does not exist before the scenario but added afterward due to its side-effect.

- Lines 36 to 67 – Specify relationships on the data. A relation is expressed with individual items, table cells and operators to be explained next.

- Line 38 – The source of an individual item is *input*, *output* or *system*. The name of an item must match one in the concrete use case.

- Line 39 – The most common operator seened in relations is equality. Since the angle brackets are used in XML to enclose element tags, they cannot be used directly. We will revisit the topic of operators shortly.

- Line 40 – A cell on a table is represented by a *table-item* element. In addition to the source, you also need to specify the table name and row number. Finally, we specify the column name of the cell which is *user-id* here.

Study the XML fragment below with scenario *passord-update.1* on page 233.

```
<scenario>
  <name id="1">password-update</name>
  <desc>change the password</desc>
  <input>
    <item value="meng">user-id</item>
    <item value="family">old-password</item>
    <item value="babygirl">new-password</item>
  </input>
  <output>
    <item value="success">result</item>
  </output>
  <system>
    <item value="2009-Feb-03">current-date</item>
    <table>
      <name>account</name>
      <column>user-id</column>
      <column>password</column>
      <column>updated-on</column>
      <row>
        <cell>francis</cell>
        <cell>hello246</cell>
        <cell>2009-Jan-25</cell>
      </row>
      <row status="old">
        <cell>meng</cell>
        <cell>family</cell>
        <cell>2009-Jan-02</cell>
      </row>
      <row status="new">
        <cell>meng</cell>
        <cell>family</cell>
        <cell>2009-Feb-03</cell>
      </row>
    </table>
  </system>
  <relation>
    <line>
      <item source="input">user-id</item>
      <op>=</op>
      <table-item source="system" table="account" row="2">user-id</table-item>
    </line>
    <line>
      <item source="input">old-password</item>
      <op>=</op>
      <table-item source="system" table="account" row="2">password</table-item>
    </line>
    <line>
      <item source="input">user-id</item>
      <op>=</op>
      <table-item source="system" table="account" row="3">user-id</table-item>
    </line>
    <line>
      <item source="input">new-password</item>
      <op>=</op>
      <table-item source="system" table="account" row="3">password</table-item>
    </line>
    <line>
      <item source="system">current-date</item>
      <op>=</op>
      <table-item source="system" table="account" row="3">updated-on</table-item>
    </line>
    <line>
      <item source="output">result</item>
      <op>=</op>
      <constant>'success'</constant>
    </line>
  </relation>
</scenario>
```

E.6.3 Validating concrete scenarios

Concrete use cases and scenarios in XML can be validated automatically. For example, we can validate a relation that equates an input item and a table cell. Given our limited resources, I have not created such a powerful validation tool. I have created an XML Schema that performs some rudimentary checks on the XML syntax of concrete use cases and concrete scenarios. For example, does a concrete scenario has a valid id number? The XML schema file used to validate our XML documents is specified on line 4 of the skeleton on next page. To do the actual validation, you need to validate the XML document from an XML authoring tool. Unless you encounter major problems writing syntactically correct concrete scenarios, I want to save time by choosing not to teach you how to use an XML authoriung tool.

Instead, I have created an XSLT stylesheet which transforms the concrete use cases and scenarios from XML to HTML. The concrete use case and its concrete scenarios are placed in the same .xml file. You then open the XML file on a Web browser. The XSLT stylesheet has been tested on Microsoft IE and Mozilla Firefox. The next page shows the skeleton of the *password-update* use case and its three concrete scenarios in an XML file. The second line is a processing instruction that names the XSLT stylesheet *concrete-scenario.xslt* to be used for the transformation.

```
<?xml version="1.0" encoding="utf-8"?>
<?xml-stylesheet type="text/xsl" href="concrete-scenario.xslt"?>
<concrete xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
          xsi:noNamespaceSchemaLocation="concrete-scenario.xsd">
  <use-case>
   .
   .
   .
  </use-case>

  <scenario>
    <name id="1">password-update</name>
    <desc>change the password</desc>
   .
   .
   .
  </scenario>

  <scenario>
    <name id="2">password-update</name>
    <desc>password update using incorrect current password</desc>
   .
   .
   .
  </scenario>

  <scenario>
    <name id="3">password-update</name>
    <desc>password update for a non-existing id</desc>
   .
   .
   .
  </scenario>
</concrete>
```

The transformation from XML to HTML allows you to visually check the correctness of concrete scenarios. It is still very far from the fully automated validation tool that we can potentially build. For instance, it does not warn you when you have mistakenly referred to a table cell using a non-existing row number. But I hope the HTML file can assist you in your manual checking.

You can open the XML files for the use cases of *create-id*, *logon* and *password-update* from a Web browser. And then open the same XML file with an editor to compare the generated HTML and original XML side by side. Lines 2 and 4 in the skeleton assume the XSD and XSLT files stored in the same directory as the XML file. Try to understand how the two files correspond to each other. I expect the specification of relations to be the area that you will most likely encounter problems. I will add a few words here. Each relation line consists of individual items, table items, constants and operators. Some operators can be specified trivially, for example, =, +, -, *, /, (and) as they display correctly in HTML. However for other operators, you will need to use the following HTML codes. Consult http://www.chami.com/tips/internet/050798I.html for operators not shown here.

Symbol	**HTML code**
≠	&#8800;
<	&#60;
≤	&#8804;
>	&#62;
≥	&#8805;

E.7 Summary

You have seen concrete use cases and their scenarios for a logon application. You have also seen the corresponding XML documents and their transformation into HTML documents for visual checking. I would like you to apply the approach to specify the functional requirements of another application. Please start thinking about some potential applications you can apply this approach. Due to the level of details and amount of work involved, you should probably start small. I would like to meet with you to discuss what application should you work on and how you should proceed.

Writing requirements specifications may not be what you are expecting. You may be more interested in writing computer programs. It is true that in the early stage of your computing career, programming abilities are important. But I must remind you that as you advance in your technical career, the ability to manage requirements becomes incrinindependenteaingly important. I believe this exposure to software requirements will benefit you.

I have not included the sequel of this tutorial because I do not want to overwhem you. The sequel involves a semi-mechanical creation of Java programs from the concrete scenarios. It also involves the creation of JUnit tests. But it is not a compulsory part of my PhD research thus has a lower priority. But you can also get involved in the programming exercise at a later time if you are interested.

If you have not done so already, please study the XML documents after opening them side-by-side in an editor and again in a Web browser. It is not necessary for you to study the XSD and XSLT files which are here to help you write correct concrete scenarios.

F. KAIN'S WORK AT THE MEETING

During our first meeting, Kain wrote the XML-based concrete use case for the borrowing of a library book as follows.

```
<use-case>
  <name>borrow a book</name>
  <desc></desc>
  <input>
    <item>user_id</item>
    <item>book_id</item>
  </input>
  <output>
    <item>result</item>
  </output>
  <system>
    <item>current-date</item>
    <table>
      <name>user</name>
      <column>user-id</column>
      <column>borrowing_quota</column>
    </table>
    <table>
      <name>borrow_record</name>
      <column>user_id</column>
      <column>book_id</column>
      <column>borrowing_date</column>
    </table>
  </system>
</use-case>
```

Kain went on to write an XML-based concrete scenario as follows.

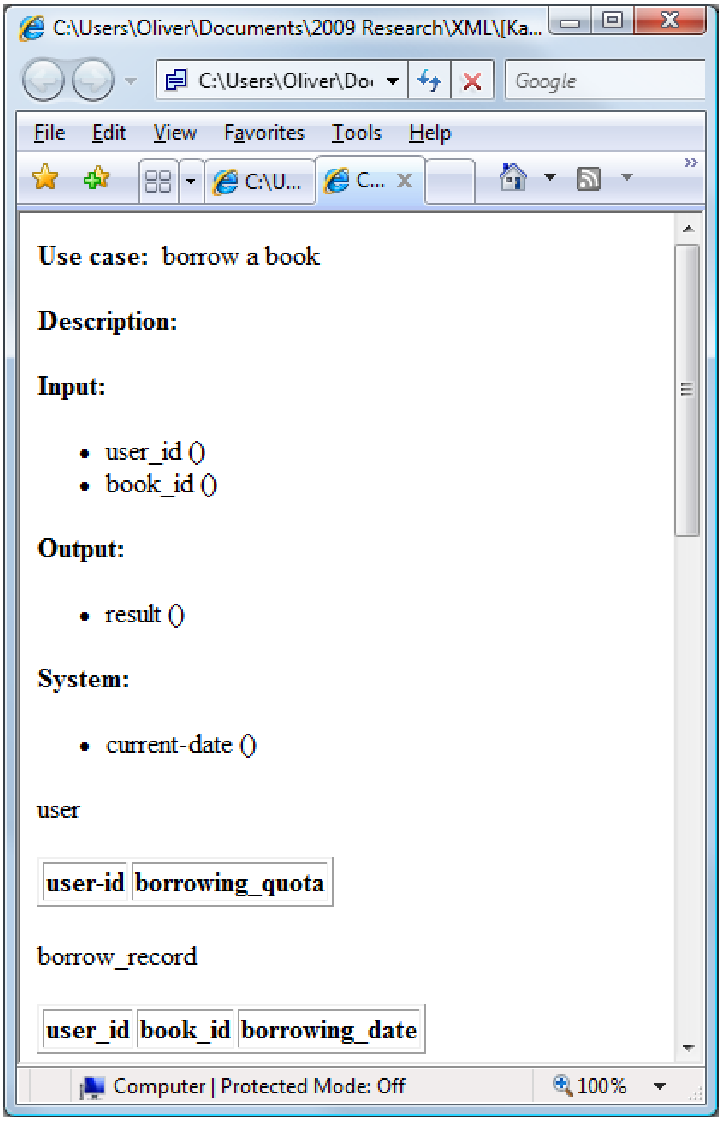
```
<scenario>
  <name id="1">borrowing</name>
  <desc>User borrow a book</desc>
  <input>
    <item value="s001">user_id</item>
    <item value="b001">book_id</item>
  </input>
  <output>
    <item value="success">result</item>
  </output>
  <system>
    <item value="2009-June-08">current-date</item>
    <table>
      <name>user</name>
      <column>user-id</column>
      <column>borrowing_quota</column>
      <row status="old">
        <cell>s001</cell>
        <cell>2</cell>
      </row>
      <row status="new">
        <cell>s001</cell>
        <cell>1</cell>
      </row>
    </table>
    <table>
      <name>borrow record</name>
      <column>user_id</column>
      <column>book_id</column>
      <column>borrowing_date</column>
      <row status="new">
        <cell>s001</cell>
        <cell>b001</cell>
        <cell>2009-6-8</cell>
      </row>
    </table>
  </system>
  <relation>
    <line>
     <item source="input">book_id</item>
     <op>=</op>
     <table-item source="system"
                 table="borrow_record" row="1">book_id</table-item>
    </line>
    <line>
     <item source="input">user_id</item>
     <op>=</op>
     <table-item source="system"
                 table="borrow_record" row="1">user_id</table-item>
    </line>
    <line>
     <item source="input">user_id</item>
     <op>=</op>
     <table-item source="system"
                 table="user" row="1">user_id</table-item>
    </line>
    <line>
     <table-item source="system"
                 table="user" row="1">borrowing_quota</table-item>
     <op>&#62;</op>
     <constant>0</constant>
    </line>
    <line>
     <item source="system">current-date</item>
     <op>=</op>
     <table-item source="system"
```

```
                table="borrow_record" row="1">borrowing_date</table-item>
    </line>
    <line>
     <table-item source="system"
                table="borrow_record" row="1">borrowing_quota</table-item>
     <op>-</op>
     <constant>1</constant>
     <op>=</op>
     <table-item source="system"
                table="borrow_record" row="2">borrowing_quota</table-item>
    </line>
    <line>
     <item source="output">result</item>
     <op>=</op>
     <constant>'Success'</constant>
    </line>
  </relation>
</scenario>
```

The concrete use case and scenarios, taking the place of the vertical dots below, are placed in the same file enclosed within the tags. The second line specifies the stylesheet file "concrete-scenario.xslt" which convert the XML to HTML for display on Web browsers. The fourth line specifies the XML schema used for syntax checking.

```
<?xml version="1.0" encoding="utf-8"?>
<?xml-stylesheet type="text/xsl" href="concrete-scenario.xslt"?>
<concrete xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
          xsi:noNamespaceSchemaLocation="concrete-scenario.xsd">
     .
     .
     .
</concrete>
```

Kain completed the above work during the meeting by using the sample XML document from version 5 of the guide as a template. The screen capture of the XML-based concrete use case and scenario displayed with our XSLT file follows.


C:\Users\Oliver\Documents\2009 Research\XML\[Ka...
C:\Users\Oliver\Do
Google
File Edit View Favorites Tools Help
C:\U...
C...
Computer | Protected Mode: Off
100%

Use case: borrow a book

Description:

Input:

- user_id ()
- book_id ()

Output:

- result ()

System:

- current-date ()

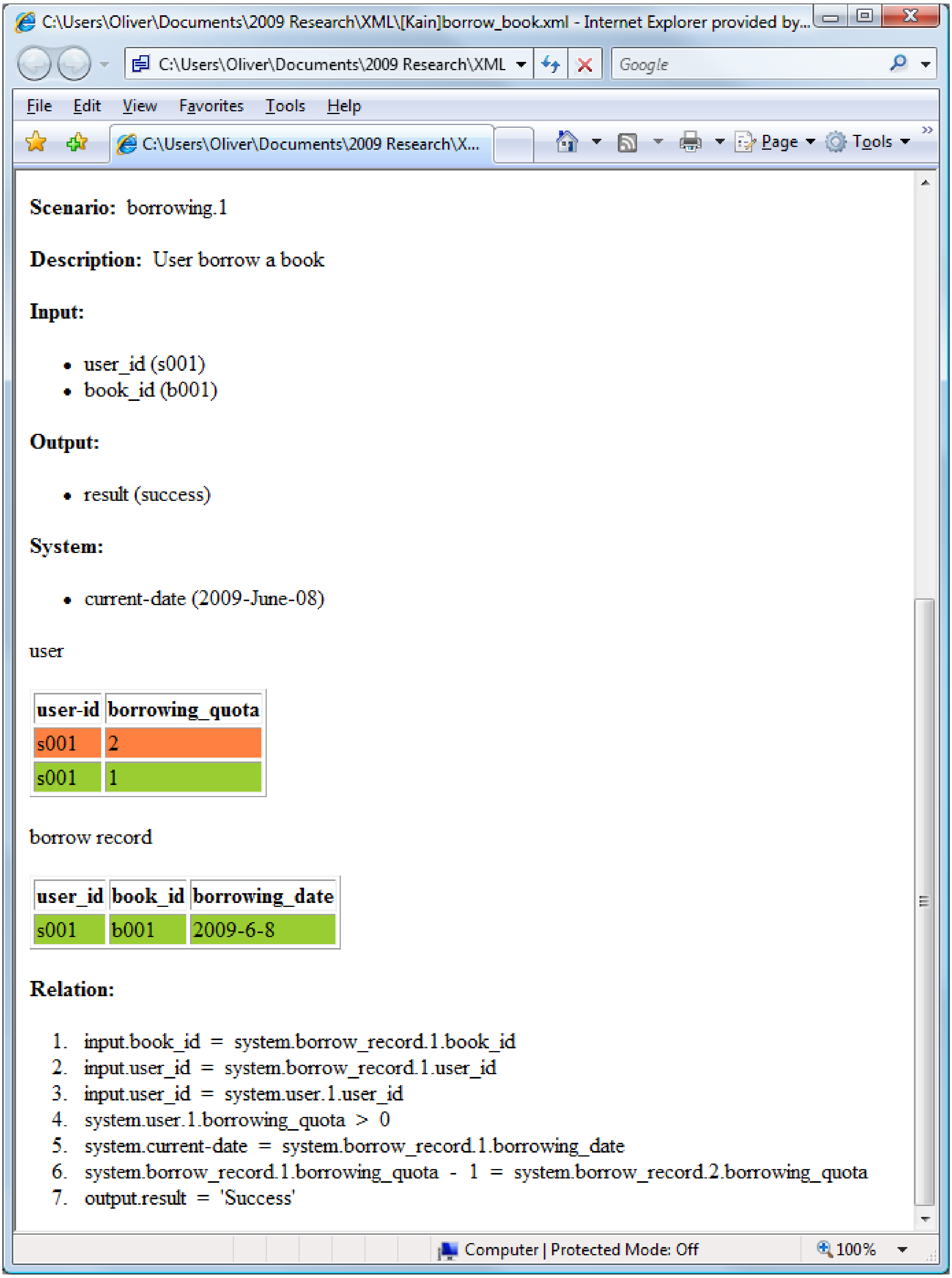
user

user-id	borrowing_quota

borrow_record

user_id	book_id	borrowing_date

Fig. F.1: Borrow a book use case by Kain


C:\Users\Oliver\Documents\2009 Research\XML\[Kain]borrow_book.xml - Internet Explorer provided by...
C:\Users\Oliver\Documents\2009 Research\XML
Google
File Edit View Favorites Tools Help
C:\Users\Oliver\Documents\2009 Research\X...
Page
Tools
Computer | Protected Mode: Off
100%

Scenario: borrowing.1

Description: User borrow a book

Input:

- user_id (s001)
- book_id (b001)

Output:

- result (success)

System:

- current-date (2009-June-08)

user

user-id	borrowing_quota
s001	2
s001	1

borrow record

user_id	book_id	borrowing_date
s001	b001	2009-6-8

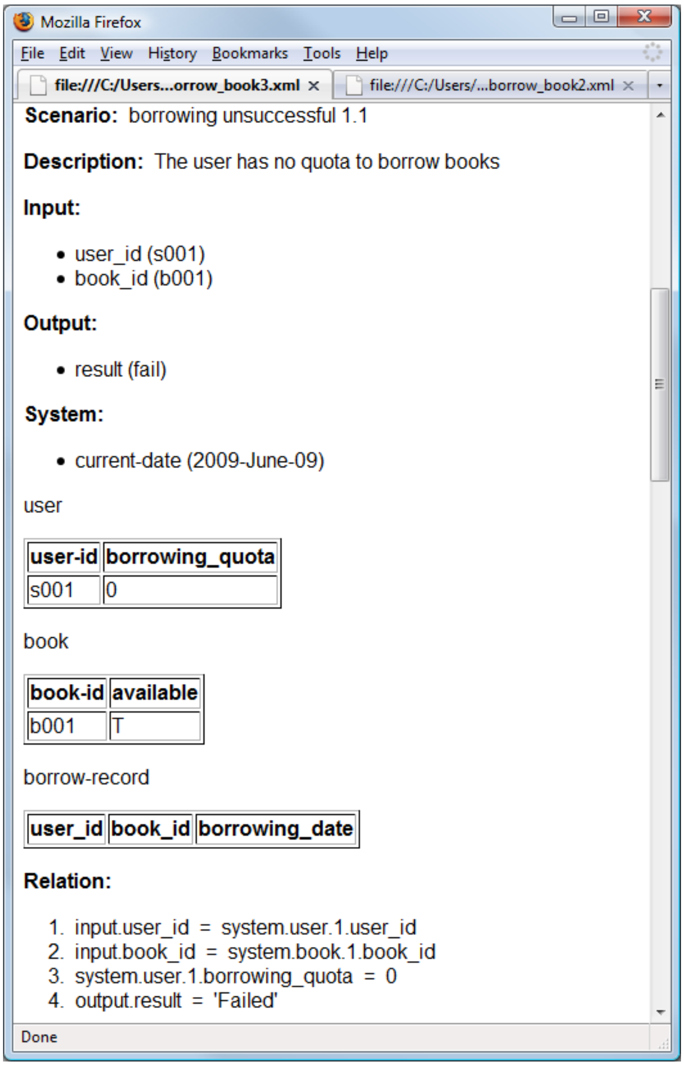
Relation:

1. input.book_id = system.borrow_record.1.book_id
2. input.user_id = system.borrow_record.1.user_id
3. input.user_id = system.user.1.user_id
4. system.user.1.borrowing_quota > 0
5. system.current-date = system.borrow_record.1.borrowing_date
6. system.borrow_record.1.borrowing_quota - 1 = system.borrow_record.2.borrowing_quota
7. output.result = 'Success'

Fig. F.2: Successful borrow a book scenario by Kain

G. LAM'S WORK

Lam created three scenarios for three different tasks shown here in the visual form.



Mozilla Firefox
File Edit View History Bookmarks Tools Help
file:///C:/Users...orrow_book3.xml
file:///C:/Users/...borrow_book2.xml
Done

Scenario: borrowing unsuccessful 1.1

Description: The user has no quota to borrow books

Input:

- user_id (s001)
- book_id (b001)

Output:

- result (fail)

System:

- current-date (2009-June-09)

user

user-id	borrowing_quota
s001	0

book

book-id	available
b001	T

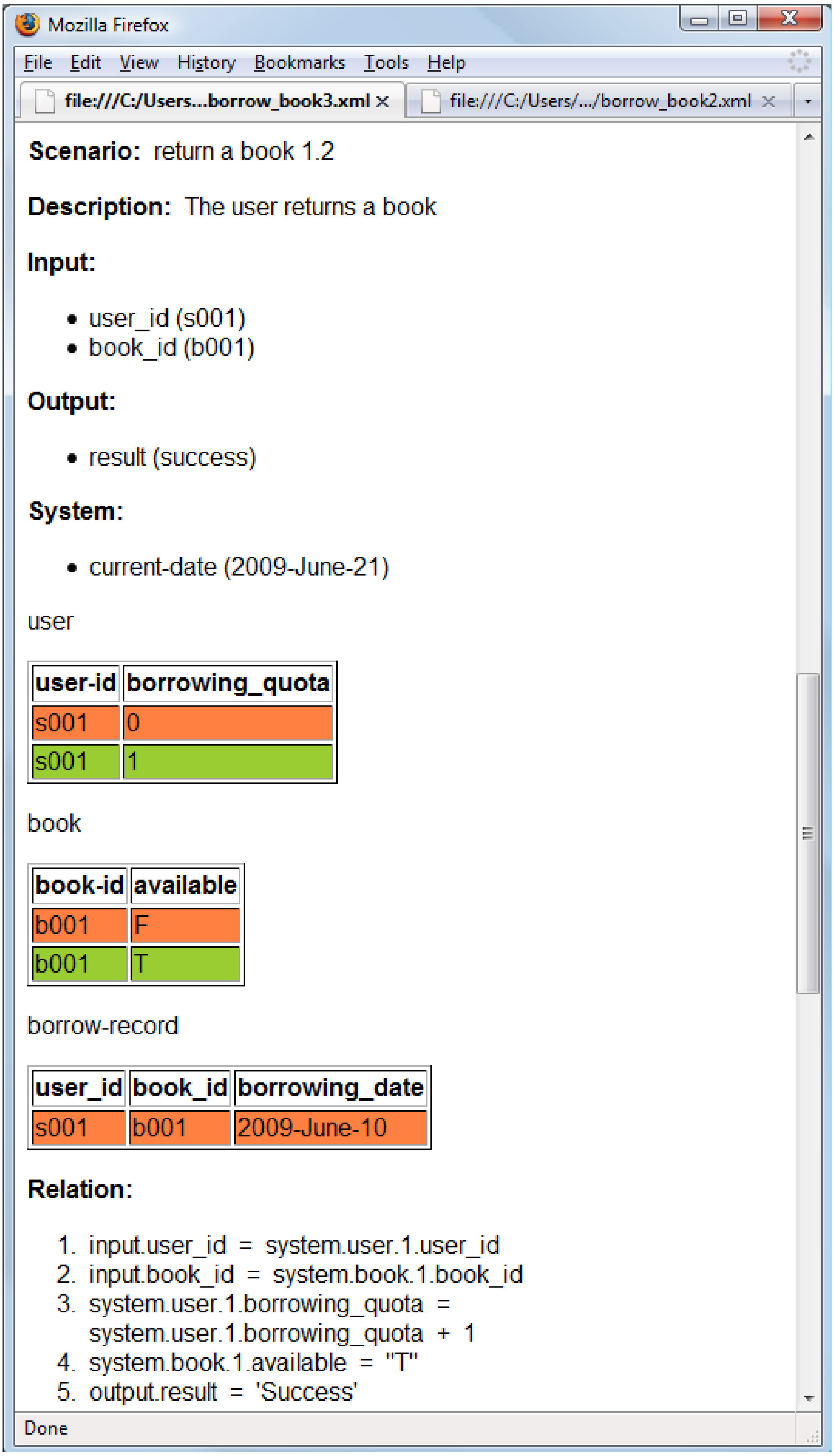
borrow-record

user_id	book_id	borrowing_date

Relation:

1. input.user_id = system.user.1.user_id
2. input.book_id = system.book.1.book_id
3. system.user.1.borrowing_quota = 0
4. output.result = 'Failed'

Fig. G.1: Unable to borrow a book by Lam


Mozilla Firefox
File Edit View History Bookmarks Tools Help
file:///C:/Users...borrow_book3.xml
file:///C:/Users/.../borrow_book2.xml
Done

Scenario: return a book 1.2

Description: The user returns a book

Input:

- user_id (s001)
- book_id (b001)

Output:

- result (success)

System:

- current-date (2009-June-21)

user

user-id	borrowing_quota
s001	0
s001	1

book

book-id	available
b001	F
b001	T

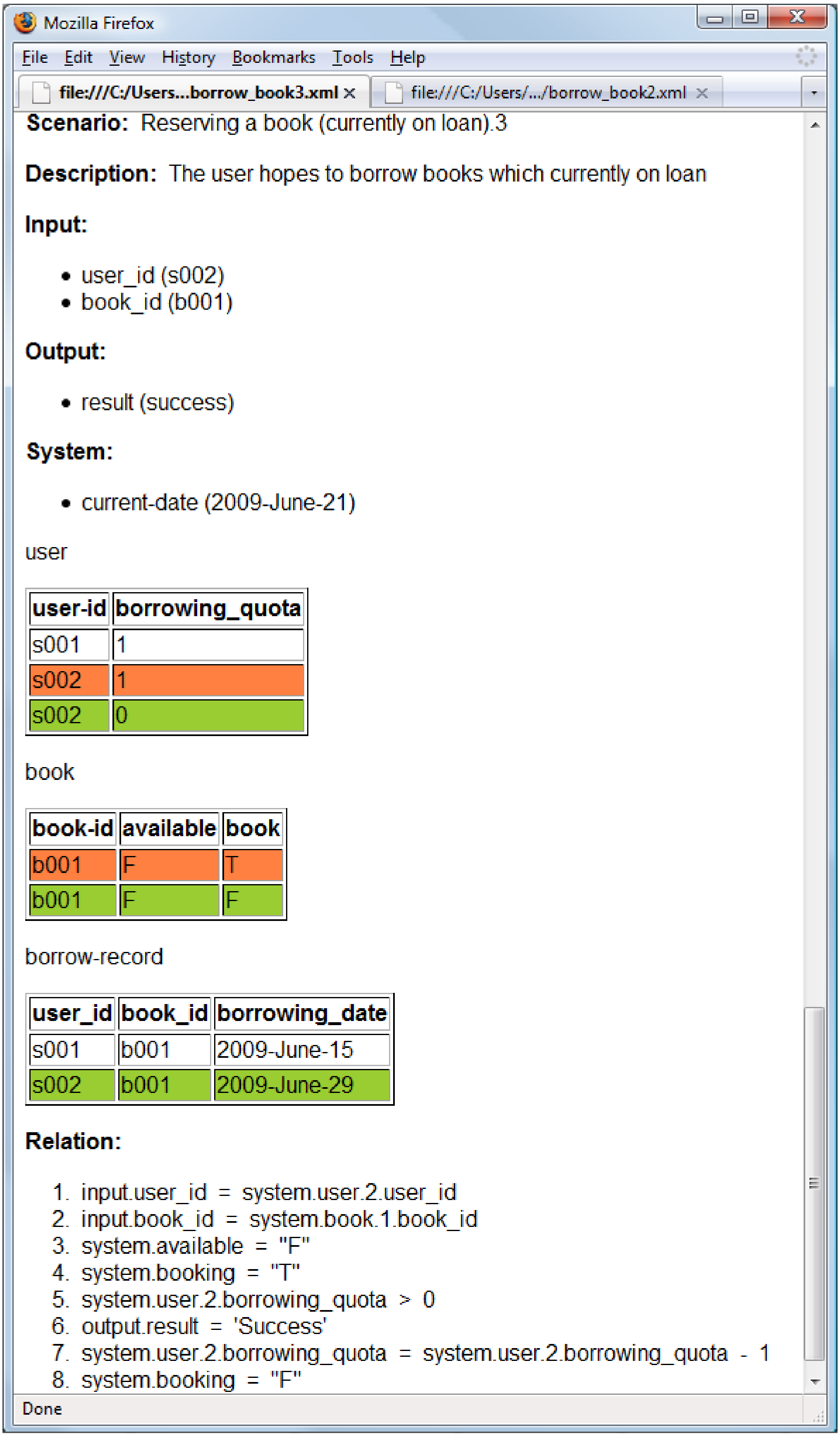
borrow-record

user_id	book_id	borrowing_date
s001	b001	2009-June-10

Relation:

1. input.user_id = system.user.1.user_id
2. input.book_id = system.book.1.book_id
3. system.user.1.borrowing_quota = system.user.1.borrowing_quota + 1
4. system.book.1.available = "T"
5. output.result = 'Success'

Fig. G.2: Returning a book by Lam



Mozilla Firefox
File Edit View History Bookmarks Tools Help
file:///C:/Users...borrow_book3.xml
file:///C:/Users/.../borrow_book2.xml
Done

Scenario: Reserving a book (currently on loan).3

Description: The user hopes to borrow books which currently on loan

Input:

- user_id (s002)
- book_id (b001)

Output:

- result (success)

System:

- current-date (2009-June-21)

user

user-id	borrowing_quota
s001	1
s002	1
s002	0

book

book-id	available	book
b001	F	T
b001	F	F

borrow-record

user_id	book_id	borrowing_date
s001	b001	2009-June-15
s002	b001	2009-June-29

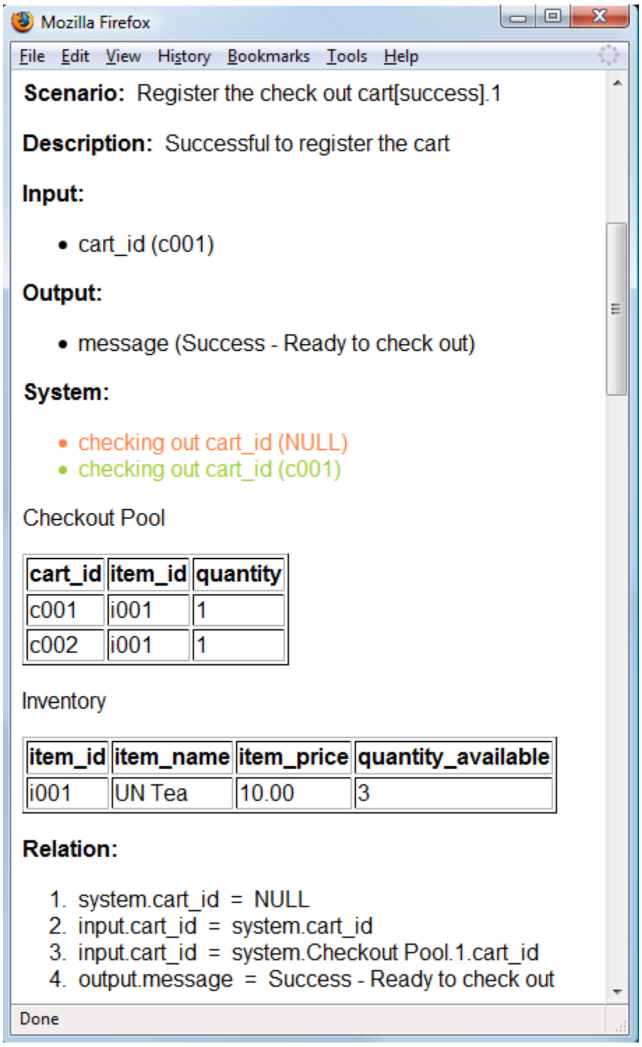
Relation:

1. input.user_id = system.user.2.user_id
2. input.book_id = system.book.1.book_id
3. system.available = "F"
4. system.booking = "T"
5. system.user.2.borrowing_quota > 0
6. output.result = 'Success'
7. system.user.2.borrowing_quota = system.user.2.borrowing_quota - 1
8. system.booking = "F"

Fig. G.3: Reserving a book by Lam

H. KAIN'S WORK

Kain created a few scenarios relating to the checking out of items in a shopping cart.



Scenario: Register the check out cart[success].1

Description: Successful to register the cart

Input:

- cart_id (c001)

Output:

- message (Success - Ready to check out)

System:

- checking out cart_id (NULL)
- checking out cart_id (c001)

Checkout Pool

cart_id	item_id	quantity
c001	i001	1
c002	i001	1

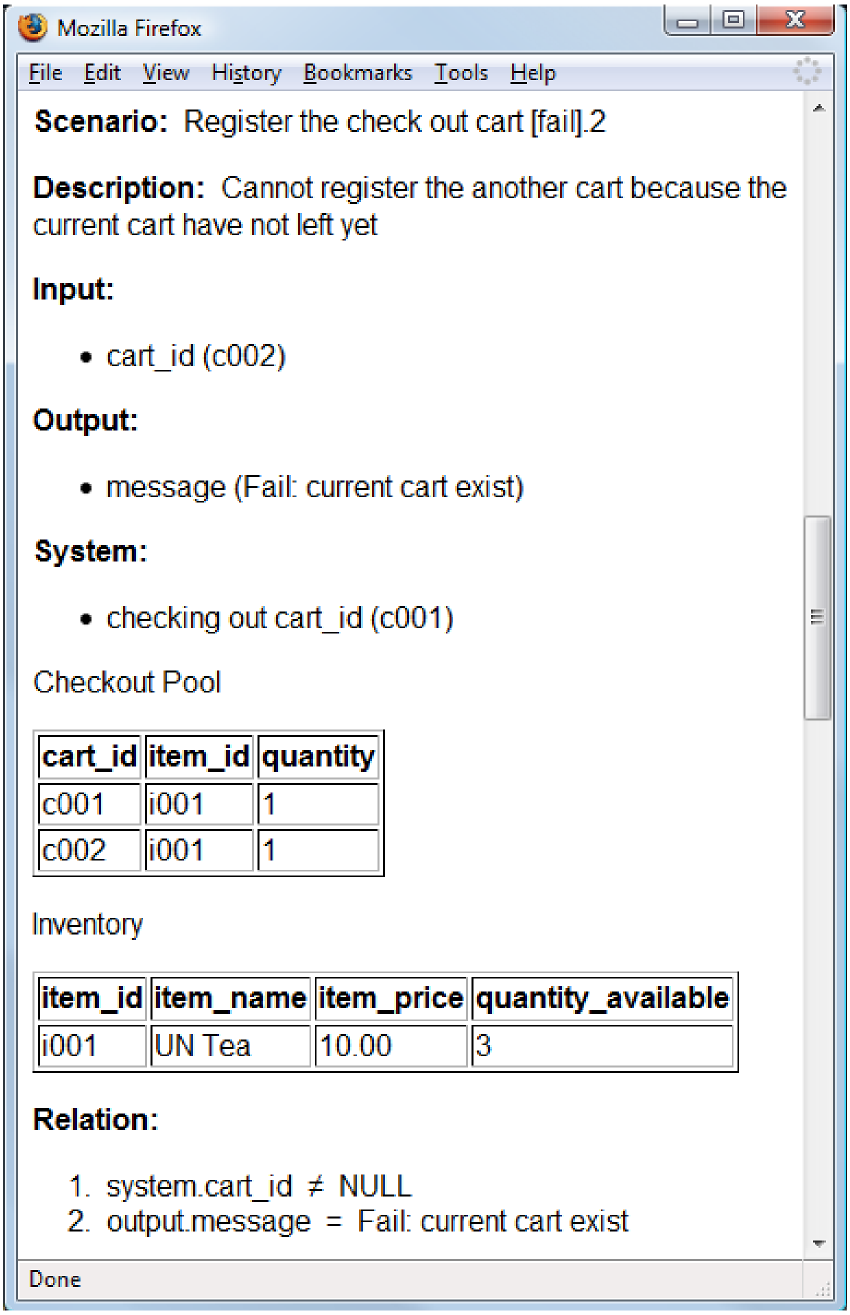
Inventory

item_id	item_name	item_price	quantity_available
i001	UN Tea	10.00	3

Relation:

1. system.cart_id = NULL
2. input.cart_id = system.cart_id
3. input.cart_id = system.Checkout Pool.1.cart_id
4. output.message = Success - Ready to check out

Fig. H.1: Push cart into unloading zone at cashier by Kain



Scenario: Register the check out cart [fail].2

Description: Cannot register the another cart because the current cart have not left yet

Input:

- cart_id (c002)

Output:

- message (Fail: current cart exist)

System:

- checking out cart_id (c001)

Checkout Pool

cart_id	item_id	quantity
c001	i001	1
c002	i001	1

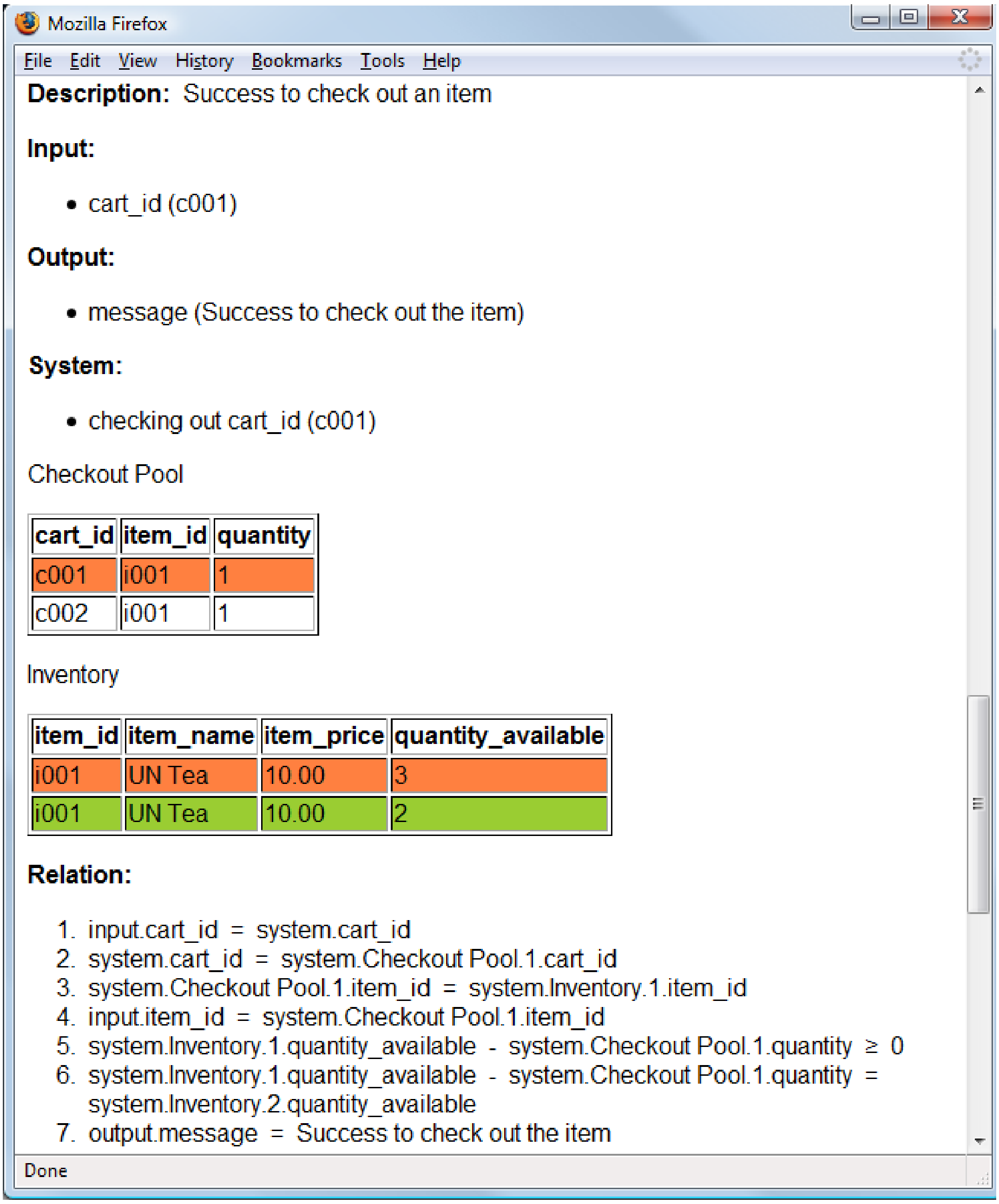
Inventory

item_id	item_name	item_price	quantity_available
i001	UN Tea	10.00	3

Relation:

1. system.cart_id ≠ NULL
2. output.message = Fail: current cart exist

Fig. H.2: Failed to push cart into an occupied unloading zone by Kain



Description: Success to check out an item

Input:

- cart_id (c001)

Output:

- message (Success to check out the item)

System:

- checking out cart_id (c001)

Checkout Pool

cart_id	item_id	quantity
c001	i001	1
c002	i001	1

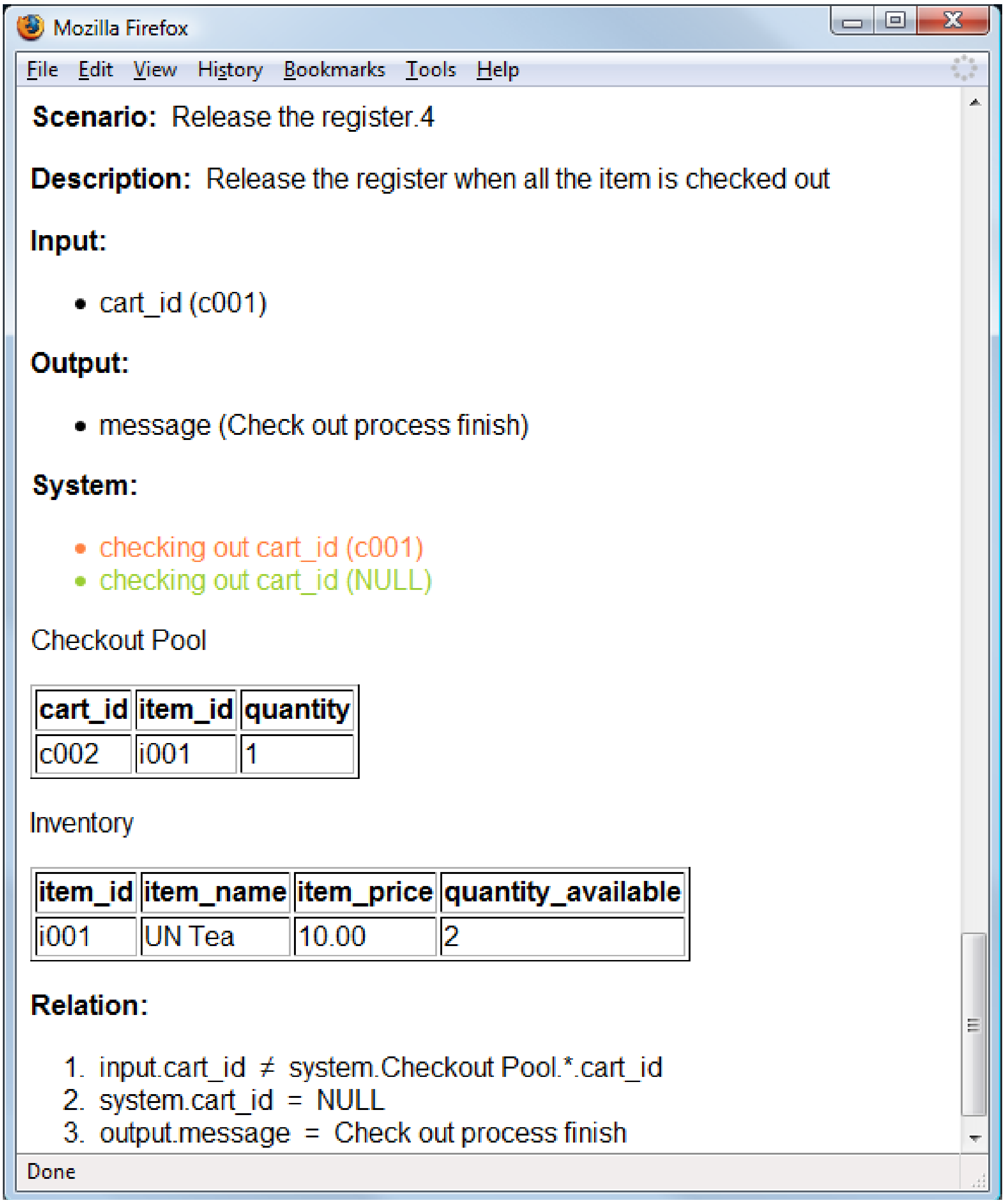
Inventory

item_id	item_name	item_price	quantity_available
i001	UN Tea	10.00	3
i001	UN Tea	10.00	2

Relation:

1. input.cart_id = system.cart_id
2. system.cart_id = system.Checkout Pool.1.cart_id
3. system.Checkout Pool.1.item_id = system.Inventory.1.item_id
4. input.item_id = system.Checkout Pool.1.item_id
5. system.Inventory.1.quantity_available - system.Checkout Pool.1.quantity ≥ 0
6. system.Inventory.1.quantity_available - system.Checkout Pool.1.quantity = system.Inventory.2.quantity_available
7. output.message = Success to check out the item

Fig. H.3: Check out an item in shopping cart by Kain



Scenario: Release the register.4

Description: Release the register when all the item is checked out

Input:

- cart_id (c001)

Output:

- message (Check out process finish)

System:

- checking out cart_id (c001)
- checking out cart_id (NULL)

Checkout Pool

cart_id	item_id	quantity
c002	i001	1

Inventory

item_id	item_name	item_price	quantity_available
i001	UN Tea	10.00	2

Relation:

1. input.cart_id $\neq$ system.Checkout Pool.*.cart_id
2. system.cart_id = NULL
3. output.message = Check out process finish

Fig. H.4: Push cart out of unloading zone at cashier by Kain

I. PROGRAM WRITING GUIDE – V1.0

Concrete scenarios have two main usages. The scenario writing guide v4.2 in Appendix C on page 188 demonstrates the first usage in which analysts express functional requirements with them. This tutorial is a sequel to the earlier guide to demonstrate the second usage in which programmers derive programs from concrete scenarios. You are expected to have studied the scenario writing guide first. Please refer back to it as necessary. We shall start with the programming of the logon task followed by the password update task.

I.1 Account Data Structure

In the logon task description on page 189, we have an account table with two fields: user id and password. Each record on the account table can be represented with a Java object. In Java's convention, we store the code of the account class in a file called *Account.java.* For the non-OO programmers out there, the construtor method is invoked when an acccount object is created. The two assignment statements inside the constructor initialise the two fields of an account object.

```
public class Account {
   String userId ;
   String password ;
   // Constructor method below has the same name as the class
   public Account (String id, String pswd) {
      userId = id ;
      password = pswd ;
   }
}
```

I.2 Logon Method

The first task we are going to build in our user management system is *logon* as described in Appendix C.2 on page 189. The task uses an account table which consists of a number of account objects. We declare the account table *acTable* as a set of account objects to save you from the distracting code used to access the database.

```
import java.util.* ;
public class UserSystem {
   // declare account table as a set of account objects
   public static Set<Account> acTable ;
   public boolean logon( String id, String pswd) { ... }
}
```

Now we are ready to define the body of the logon method. The relation components of the three logon scenarios from page 190 to page 191 the scenario writing guide are reproduced below.

logon.1	input.user-id = account.2.user-id
	input.password = account.2.password
logon.2	input.user-id = account.1.user-id
	input.password ≠ account.1.password
logon.3	input.user-id ≠ account.1.user-id
	input.user-id ≠ account.2.user-id
	input.user-id ≠ account.3.user-id

The matching of the input id and password in scenario *logon.1* can be implemented with the conditions in two if-statements. Though the scenario shows a match on the second row of the data table, the match can be generalised to any row implemented by a while-loop that iterates through every account object. Scenario *logon.2* differs from scenario *logon.1* only in the unmatched password. This is implemented conveniently by the else-branch of the second if-statement. Scenario *logon.3* allows no match for the input id on any row in the table. This is implemented with a trival return outside the while-loop. The resulting program is shown next.

```
   import java.util.* ;
   public class UserSystem {
      // declare account table as a set of account objects
      public static Set<Account> acTable ;
      public boolean logon( String id, String pswd) {
         Iterator iterator = acTable.iterator( ) ;
         while( iterator.hasNext( ) ) {
            Account ac = (Account) iterator.next( ) ;
            if ( ac.userId.equals(id) ) {
               if ( ac.password.equals(pswd) )
                  return true ;      // scenario logon.1
               else
                  return false ;     // scenario logon.2
            }
         };
         return false ;              // scenario logon.3
      }
}
```

We implement a user task with a method. The input parameters of the logon task, user id and password, become the method's two arguments. The task result of success or failure is captured by the method's boolean return type. The task's input and output parameters are fully covered by the method's signature.

The logon method shows the use of an *iterator* stepping through the set of account objects. The method *hasNext*( ) tests if there is at least one object left in the iterator. The method *next*( ) actually returns the next object. But the returned oject is a generic one which must be recasted to an account object before being assigned to the variable *ac*. Scenario *logon.3* requires none of the exising account names to match the input user id. Therefore its corresponding statement to return a failed result is placed outside the while-loop after all account names have been exhausted. The *equals*( ) method used in the if-statements compares two strings.

I.3 Logon Tests

After we have written the logon method for our user management system, we create a test suite capturing the concrete scenarios in JUnit which is a framework created by Kent Beck to facilitate automated testing. Beck is also the inventor of eXtreme Programming (XP).

```
import org.junit.* ;
import static org.junit.Assert.* ;
public class LogonTest {
   UserSystem mySystem = new UserSystem() ;
   @Before
   public void initialiseTable() {
      System.out.print("Initialise User System -- ") ;
      mySystem.acTable = new HashSet<Account>( ) ;
      mySystem.acTable.add( new Account("cudie", "7there59"));
      mySystem.acTable.add( new Account("francis", "hello246"));
      mySystem.acTable.add( new Account("meng", "family"));
   }
   @Test
   public void test_logon_1() {
      System.out.println("Scenario logon.1") ;
      assertTrue(mySystem.logon("francis", "hello246")) ;
   }
   @Test
   public void test_logon_2() {
      System.out.println("Scenario logon.2") ;
      assertFalse(mySystem.logon("francis", "goodbye")) ;
   }
   @Test
   public void test_logon_3() {
      System.out.println("Scenario logon.3") ;
      assertFalse(mySystem.logon("jasper", "alberta")) ;
   }
}
```

We name our test suite *LogonTest*. The test suite was written in the same style as explicated in a JUnit tutorial [118]. The *LogonTest* class uses an instance of *UserSystem* defined on page 261. The suite has three tests: *test_logon_1*( ), *test_logon_2*( ) and *test_logon_3*( ). JUnit does not care what we call the tests as long as they are appropriately annotated with '@Test'. In each test, we invoke the *logon* method with the input arguments from the corresponding scenario. Methods *assertTrue* and *assertFalse* are provided by JUnit to test boolean expressions. Since scenario *logon.1* is expected to return true, we call *assertTrue*. The other two scenarios are expected to return false, we call *assertFalse*.

Method *initialiseTable* is annotated with '@Before'. It is invoked automatically before each test to create three ids on the account table: *cudie*, *francis* and *meng*.

I.4 Running Logon Tests

JUnit can be used with advanced IDE's such as Eclipse. However we make the minimal assumptions about readers' Java background and installations. The tests will be executed from the MS Windows command prompt.

I.4.1 Downloading JDK

From http://java.sun.com/javase/, find a link to download and install Java Standard Edition (Java SE) Development Kit (JDK). The current version at the time of writing installs files in folder c:\Program Files\Java\jdk1.6.0_06.

I.4.2 Setting the Path

Path is an environment variable used to find the program names you enter on the command prompt. You can enter *path* to see the current list of directories from which MS Windows tries to find programs. You can append the path of the JDK to the end of the path list with the following command.

```
set path=%path%;C:\Program Files\Java\jdk1.6.0_06\bin
```

I.4.3 Getting All Files

You need the following files to run the test suite.

1. Account.java
2. UserSystem.java
3. LogonTest.java
4. junit-4.5.jar

The first three are Java source files obtainable from us. The last file is the current version 4.5 of JUnit downloadable from http://www.junit.org/. Its file type *jar* is the Java archive file format that aggregates multiple files into one.

I.4.4 Compilation and Execution

After you have stored the four files in a single folder of your choice. You compile them with the following commands.

```
javac Account.java
javac UserSystem.java
javac -cp .;junit-4.5.jar LogonTest.java
```

Finally, you can run the test suite.

```
java -cp .;junit-4.5.jar org.junit.runner.JUnitCore LogonTest
```

You will get the following expected result for three tests. Three lines were output by the print statements we inserted in the test suite.

```
JUnit version 4.5
.Initialise User System -- Scenario logon.1
.Initialise User System -- Scenario logon.2
.Initialise User System -- Scenario logon.3

Time: 0.048

OK (3 tests)
```

Suppose by mistake we have coded *assertFalse*( ) in place of *assertTrue*( ) in the third test *test_logon_3*. When we recompile and rerun *LogonTest*, JUnit will identify the failed test method followed by a dump of the current call stack.

```
JUnit version 4.5
.Initialise User System -- Scenario logon.1
.Initialise User System -- Scenario logon.2
.Initialise User System -- Scenario logon.3
E
Time: 0.053
There was 1 failure:
1) test_logon_3(LogonTest)
java.lang.AssertionError:
     at org.junit.Assert.fail(Assert.java:91)
     at org.junit.Assert.assertTrue(Assert.java:43)
     at org.junit.Assert.assertTrue(Assert.java:54)
     at LogonTest.test_logon_3(LogonTest.java:33)
      .
      .
      .
FAILURES!!!
Tests run: 3,  Failures: 1
```

I.5 Password Update Method

The next task we are going to build in our user management system is *password-update* as described in Appendix C.3 on page 191. The task shares the same account table with the logon task. We do not need to create new data structure. The relations of the three password update scenarios are reproduced below.

password-update.1 input.user-id = account.1.user-id
input.old-password = account.1.password
input.user-id = account.2.user-id

	input.new-password = account.2.password
password-update.2	input.user-id = account.1.user-id
	input.old-password ≠ account.1.password
password-update.3	input.user-id ≠ account.1.user-id
	input.user-id ≠ account.2.user-id

The password update task is implemented with a method in the user system. The new method has three arguments to match the task's input parameters. The successive deletion and addition in the scenario is more directly implemented by an update to the password field. Other than the new input parameter and the side-effect on the account data, the password update method is very similar to the logon method.

```
   import java.util.* ;
   public class UserSystem {
      public static Set<Account> acTable ;
      public boolean logon( String id, String pswd) { ... }
      public boolean passwordUpdate( String id,
                                     String oldPswd,
                                     String newPswd) {
         Iterator iterator = acTable.iterator( ) ;
         while(iterator.hasNext( )) {
            Account ac = (Account) iterator.next( ) ;
            if ( ac.userId.equals(id) ) {
               if ( ac.password.equals(oldPswd) ) {
                  ac.password = newPswd ;  // scenario password-update.1
                  return true ;
               }
            else
               return false ;              // scenario password-update.2
         };
         return false ;                    // scenario password-update.3
      }
}
```

I.6 Password Update Tests

```
import java.util.* ;
import org.junit.* ;
import static org.junit.Assert.* ;

public class PasswordUpdateTest {
   UserSystem mySystem = new UserSystem() ;
   @Before
   public void initialiseTable() {
      mySystem.acTable = new HashSet<Account>( ) ;
      mySystem.acTable.add( new Account("francis", "hello246"));
      mySystem.acTable.add( new Account("meng", "family"));
   }

   @Test
   public void test_password_update_1() {
      assertTrue(mySystem.passwordUpdate("meng", "family", "babygirl")) ;
      assertFalse(mySystem.logon("meng", "family")) ;     // old password
      assertTrue(mySystem.logon("meng", "babygirl")) ;    // new password
      assertTrue(mySystem.logon("francis", "hello246")) ; // current password
   }
   @Test
   public void test_password_update_2() {
      assertFalse(mySystem.passwordUpdate("francis", "goodbye", "goodwill")) ;
   }
   @Test
   public void test_password_update_3() {
      assertFalse(mySystem.passwordUpdate("jasper", "calgary", "edmonton")) ;
   }
}
```

The three scenarios are encoded in three test methods. In the first method, we make additional calls to the logon method to more thoroughly test the user system after a successful password update operation.

```
javac UserSystem.java
javac -cp .;junit-4.5.jar PasswordUpdateTest.java
java -cp .;junit-4.5.jar org.junit.runner.JUnitCore PasswordUpdateTest
```

The above commands compile and execute the password update tests to give the following result. The output is cleaner than before because we have not inserted print statements in the tests. You see three dots between the JUnit version number and the elapsed time, one for a test method. When we have dozens of more involved test methods in the suite, the expanding line of dots informs us that the testing is still alive.

```
JUnit version 4.5
...
Time: 0.041

OK (3 tests)
```

I.7 Summary

We shall recap the steps to derive Java programs from concrete scenarios. The coding is performed task by task. If you are new to writing programs from concrete scenarios, I would suggest you to begin with a simple task using only simple data structure. After you have become proficient in the approach, you may start with tasks considered important or urgent by the customers.

We will look at the data manipulated by the task to create classes for data objects and tables. We will then create a method for each task. The method must have input and output parameters to match the task.

We go on to write the code for the method. We need to look at all the scenarios of the task especially their relation components. The relations refer to specific data values. You will generalise them to program statements expressed with variables. Often two branches of an if-statement implement two scenarios. We can also deal with one scenario inside a loop and another scenario outside the loop.

Sometimes the analyst may have overlooked a scenario. That generally will result in a dangling else-branch or unspecific action after exiting a loop. The programmer can confirm with the anlayst for possible omissions. For example, the analyst may have omitted scenario *logon.3*. The programmer will generalise scenarios *logon.1* and *logon.2* to the following code which does not specify what to return after the while loop has terminated. The programmer can bring it to the attention of the analyst.

```
import java.util.* ;
public class UserSystem {
   public static Set<Account> acTable ;
   public boolean logon( String id, String pswd) {
      Iterator iterator = acTable.iterator( ) ;
      while( iterator.hasNext( ) ) {
         Account ac = (Account) iterator.next( ) ;
         if ( ac.userId.equals(id) ) {
            if ( ac.password.equals(pswd) )
               return true ;      // scenario logon.1
            else
               return false ;     // scenario logon.2
         }
      };
      // What to return at this point? Missing a scenario?
   }
}
```

GLOSSARY

Black box testing The test designer assumes no knowledge of how the module being tested is implemented. Therefore black box tests are limited to the checking of correct output for specific input.

Concrete Scenarios A concrete scenario describes an operation of a system with an actual example. The description includes the actual values used in the input and output parameters and the system states before and after the operation. System states are represented by individual data fields and rows of data in tables. Data fields and rows are instantiated to specific values not ranges of values. Many concrete scenarios in this thesis are described with two states: a begin state and an end state. Some concrete scenarios are described with intermediate states representing additional decisions made by the analysts. The syntax of concrete scenarios has evolved over the course of our research. The concrete scenarios in Appendix E uses the latest syntax that aims to facilitate the deriviation of programs by programmers other than the researcher.

Customer Scenarios Customer scenarios are concrete scenarios which express examples to the level of details cared by the customers. The meaning of the term *customers* is taken broadly. In an online bookshop development project, customers include the business manager, the staff who operate the website on a daily basis, shoppers and users who are just browsing. Table 7.1 on page 115 is a customer scenario. The scenario shows a sorted list as the outcome. It does not show an algorithm which the customer does not care.

Developer Scenarios A developer scenario is built on top of a customer scenario. A developer scenario has details to help the developer to visualize how the customer scenario may be implemented. There may be new data fields and intermediate states to capture progress to the desirable ending state. Tables 7.5 and 7.6 show two possible developer scenarios based on the same customer scenario in Table 7.4. Essentially a developer scenario adds algorithm-related information to the customer scenario.

E-Scenarios An E-scenario is a form of concrete scenarios where data fields are embedded in English sentences to give them meaning in the application domain. Data rows from the same table will follow the same sentence template. Section 3.2.2 on page 51 has a detailed explanation. Figure 1.2 on Page 6 shows how they fit in the software development process.

Expansion Expansion is the manual process to create developer scenarios from customer scenarios. A choice of algorithm is made and is represented with actual data. This is an optional process that brings concrete scenarios closer to implementations. Programmers can derive programs directly from customer scenarios or indirectly through developer scenarios. The process allows programmers to grasp complex computations with examples. See Chapter 7 for details.

Generalization This is a task for programmers and formal specification writers. It turns the specific data relationships in concrete scenarios to executable statements in programs or logical expressions in formal specifications. Data values are replaced by variables. See Figure 1.2 on Page 6 and explanations on Page 50.

Observance Observance is a formally defined relationship between a set of Z operation schemas and a set of concrete scenarios. The formal foundation is defined using value substitions. If all logical expressions in Z schemas evaluate to true after substitions specified in scenarios, we know that our specification in Z or other formalism satisfies the requirements expressed in the scenarios. See details in Chapter 6.

White box testing It utilises internal knowledge of the module being tested.

For example, path testing is a kind of white box testing that covers all possible paths of execution through this module.

Z-Scenarios A Z-scenario is a form of concrete scenarios. There is a one-to-one correspondence between E-scenarios and Z-scenarios. Sentences from E-scenarios are rewritten in Z notation. Data fields from the same sentence in E-scenario are related in a Z maplet $\mapsto$. See Table 3.6 on page 55 for an example.

BIBLIOGRAPHY

[1] *i** wiki. http://istar.rwth-aachen.de/.

[2] J. R. Abrial. *The B-Book : Assigning Programs to Meanings.* Cambridge University Press, 1996.

[3] V. S. Alagar and K. Periyasamy. *Specification of Software Systems.* Springer-Verlag, 1998.

[4] T. A. Alspaugh, S. R. Faulk, K. H. Britton, R. A. Parker, D. L. Parnas, and J. E. Shore. Software requirements for the A-7E aircraft. Technical Report NRL-9194, Naval Research Laboratory, Washington, DC, 1992.

[5] Scott W. Ambler. UML2 sequence diagrams. http://www.agilemodeling.com/artifacts/sequenceDiagram.htm, April 2006.

[6] Daniel Amyot. ITU-T's user requirements notation (URN) and jUCMNav. In *CASCON'08 – Requirements-Driven Business Process Modeling and Performance Management.* IBM Press, October 2008.

[7] Oliver Au, Roger Stone, and John Cooke. Precise scenarios – a customer-friendly foundation for formal specifications. In *iFM: integrated Formal Methods*, pages 21–36, Oxford, United Kingdom, July 2007. Springer-Verlag.

[8] M. Autili, P. Inverardi, and P. Pelliccione. A scenario based notation for specifying temporal properties. In *The international workshop on Scenarios and state machines: models, algorithms, and tools*, pages 21–28, Shanghai, China, May 2006. Springer-Verlag.

[9] R. J. R. Back and R. Kurki-Suonio. Distributed cooperation with action systems. *ACM Transactions on Programming Languages and Systems*, 10(4):513–554, October 1988.

[10] Ralph-Johan Back and Joakim von Wright. *Refinement Calculus - A Systematic Introduction.* Springer, 1998.

[11] J. L. Baer. A survey of some theoretical aspects of multiprocessing. *ACM Computing Surveys*, 5(1), March 1973.

[12] Marko Bajec and Marjan Krisper. A methodology and tool support for managing rules in organisations. *Information Systems*, 30(6):423–443, September 2005.

[13] Hubert Baumeister. Combining formal specifications with test driven development. In *XP/Agile Universe*, volume 3134 of *Lecture Notes in Computer Science*, pages 1–12, Calgary, Canada, August 2004. Springer-Verlag.

[14] Kent Beck. *Extreme Programming Explained: Embrace Change.* Addison Wesley, 2000.

[15] Kent Beck. *Test-Driven Development: By Example.* Addison Wesley, 2003.

[16] P. Bellini, R. Mattolini, and P. Nesi. Temporal logics for real-time system specification. *ACM Computing Surveys*, 32(1), March 2000.

[17] Luca Bernardinello and Fiorella de Cindio. A survey of basic net models and modular net classes. In *Advances in Petri Nets 1992, The DEMON Project*, pages 304–351, London, UK, 1992. Springer-Verlag.

[18] Dines Bjørner. Formal methods in the 21'st century: An assessment of today – predictions for the future. In *Proc. ICSE'98*, 1998.

[19] Andreas Blass and Yuri Gurevich. Abstract state machines capture parallel algorithms: Correction and extension. *ACM Transactions on Computational Logic*, pages 1–29, January 2007.

[20] Barry W. Boehm. *Software Engineering Economics.* Prentice Hall, 1981.

[21] Barry W. Boehm. A spiral model of software development and enhancement. *IEEE Computer*, 21(5):61–72, 1988.

[22] R. J. Boland. Phenomenology: A preferred approach to research in information systems. In E. Mumford, R.A. Hirschheim, G. Fitzgerald, and T. WoodHarper, editors, *Research Methods in Information Systems*, pages 193–201. NorthHolland, 1985.

[23] Grady Booch, James Rumbaugh, and Ivar Jacobson. *The Unified Modeling Language User Guide.* Addison-Wesley, 1999.

[24] Egon Börger, Elvinia Riccobene, and Joachim Schmid. Capturing requirements by abstract state machines: The light control case study. *Journal of Universal Computer Science*, 6(7):597–620, 2000.

[25] Jonathan P. Bowen and Michael G. Hinchey. Seven more myths of formal methods. *IEEE Software*, pages 34–41, July 1994.

[26] M. Brandozzi and D. E. Perry. Transforming goal oriented requirement specifications into architecture prescriptions. In *STRAW'03: Second International Workshop from Software Requirements to Architectures*, Portland, Oregon, May 2003. IEEE Press.

[27] Monica Brockmeyer. Using modechart modules for testing formal specifications. In *4th IEEE International Symposium on High-Assurance Systems Engineering*, 1999.

[28] S. E. Bronner and D. M. Kellner, editors. *Critical Theory and Society: a Reader.* Taylor & Francis, 1989.

[29] Lee Brownston, Robert Farrell, Elaine Kant, and Nancy Martin. *Programming Expert Systems in OPS5: An Introduction to Rule-Based Programming.* Addison-Wesley, 1985.

[30] Manfred Broy. Formal description techniques - how formal and descriptive are they? In R. Gotzhein and J. Bredereke, editors, *Formal*

Description Techniques IX, Theory, Application and Tools, pages 95–110. Chapman and Hall, 1996.

[31] François Bry, Michael Eckert, Paula-Lavinia Patranjan, and Inna Romanenko. Realizing business processes with ECA rules: Benefits, challenges, limits. In José Júlio Alferes, James Bailey, Wolfgang May, and Uta Schwertel, editors, *Principles and Practice of Semantic Web Reasoning, 4th International Workshop, PPSWR 2006, Revised Selected Papers*, volume 4187 of *Lecture Notes in Computer Science*, pages 48–62, Budva, Montenegro, June 2006. Springer.

[32] R. B. Burns. *Introduction to Research Methods.* SAGE Publications, 2000.

[33] D. W. Bustard and A. C. Winstanley. Making changes to formal specifications: Requirements and an example. *IEEE Trans. Softw. Eng.*, 20(8):562–568, 1994.

[34] J. Castro, M. Kolp, and J. Mylopoulous. Towards requirements-driven information systems engineering: the tropos project. *Information Systems*, 27:365–389, 2002.

[35] K. Mani Chandy and Jayadev Misra. *Parallel Program Design: A Foundation.* Addison-Wesley, 1989.

[36] Marsha Chechik. SC(R)^3: Towards usability of formal methods. In *CASCON'98*, pages 177–191. IBM Press, November 1998.

[37] L. Chung, B. A. Nixon, E. Yu, and J. Mylopoulous. *Non-functional Requirements in Software Engineering.* Kluwer (Springer), 1999.

[38] Alistair Cockburn. *Writing Effective Use Cases.* Addison-Wesley, 2001.

[39] John Cooke. *Constructing Correct Software.* Springer, second edition, 2005.

[40] P.-J. Courtois and D. L. Parnas. Documentation for safety critical software. In *ICSE'93: 15th International Conference on Software Engineering*, pages 315–323, 1993.

[41] Ward Cunningham. Fit: Framework for integrated test. http://fit.c2.com/, January 2007.

[42] Alan M. Davis. *Software Requirements Analysis and Specification.* Prentice Hall, 1990.

[43] Alan Dennis, Barb Wixom, and Robby Roth. *Systems Analysis & Design.* John Wiley & Sons, fourth edition, 2009.

[44] R. Diaconescu and K. Futatsugi, editors. *CafeOBJ Report: The Language, Proof Techniques, and Methodologies for Object-Oriented Algebraic Specification.* World Scientific, 1998.

[45] Edsger W. Dijkstra. Notes on structured programming. Technical Report 70-WSK-03, Technological University Department of Mathematics, Einhoven, The Netherlands, April 1970.

[46] Edsger W. Dijkstra, editor. *A Discipline of Programming.* Prentice Hall, 1976.

[47] J. Douglas and R.A. Kemmerer. Aslantest: A symbolic execution tool for testing Aslan formal specifications. In *Proceedings of the 1994 ACM SIGSOFT International Symposium on Software Testing and Analysis*, pages 15–27. ACM Press, 1994.

[48] Alexander Egyed, Paul Grünbacher, and Nenad Medvidovic. Refinement and evolution issues in bridging requirements and architecture vthe cbsp approach. In *STRAW'01: First International Workshop from Software Requirements to Architectures*, Toronto, May 2001. IEEE Press.

[49] Albert Endres. A synopsis of software engineering history: The industrial perspective. In Andreas Brennecke and Reinhard Keil-Slawik, editors, *Position Papers for Dagstuhl Seminar 9635 on History of Software Engineering*, pages 20–24, August 1996.

[50] S. R. Faulk, L. Finneran, J. Kirby JR., S. Shah, and J. Sutton. Experience applying the core method to the lockheed C-130J. In *COM-*

PASS'94: 9th Annual Conference on Computer Assurance, pages 3–6, June 1994.

[51] Kate Finney. Mathematics notation in formal specification: Too difficult for the masses? *IEEE Trans. Softw. Eng.*, 22(2):158–159, 1996.

[52] Kate M. Finney and Alex M. Fedoree. An empirical study of specification readability. In C. Neville Dean and Michael G. Hinchey, editors, *Teaching and Learning Formal Methods*, pages 117–129. Academic Press, 1996.

[53] Martin Fowler. *Refactoring: Improving the Design of Existing Code.* Addison Wesley, 1999.

[54] R. D. Franklin, D. B. Allison, and B. S. Gorman. *Design and Analysis of Single-Case Research.* Lawrence Erlbaum, 1996.

[55] Martin D. Fraser, Kuldeep Kumar, and Vijay K. Vaishnavi. Informal and formal requirements specification languages: Bridging the gap. *IEEE Transactions on Software Engineering*, 17(5), May 1991.

[56] Norbert E. Fuchs, Uta Schwertel, and Rolf Schwitter. Attempto Controlled English — not just another logic specification language. *Lecture Notes in Computer Science*, 1559:1–20, 1999.

[57] Vincenzo Gervasi. Synthesizing ASMs from natural language requirements. In *the 8th EUROCAST Workshop on Abstract State Machines*, pages 212–215, 2001.

[58] Carlo Ghezzi, Mehdi Jayazeri, and Dino Mandrioli. *Fundamentals of Software Engineering.* Prentice Hall, 1991.

[59] Robert Glass. The mystery of formal methods disuse. *Communication of the ACM*, 47(8), August 2004.

[60] Daniel Gross and Eric Yu. Evolving system architecture to meet changing business goals: an agent and goal-oriented approach. In *STRAW'01: First International Workshop from Software Requirements to Architectures*, Toronto, May 2001. IEEE Press.

[61] The Standish Group. The chaos report. http://www.standishgroup.com/sample_research/chaos_1994_1.php, 1994.

[62] Yuri Gurevich. Logic and the challenge of computer science. In Egon Börger, editor, *Current Trends in Theoretical Computer Science*, pages 1–57. Computer Science Press, 1988.

[63] Yuri Gurevich. May 1997 draft of the asm guide. Technical Report CSE-TR-336-97, University of Michigan EECS Department, May 1997.

[64] Yuri Gurevich. Sequential abstract state machines capture sequential algorithms. *ACM Transactions on Computational Logic*, 1(1):77–111, July 2000.

[65] J. V. Guttag, J. J. Horning, S. J. Garland, K. D. Jones, A. Modet, and J. M. Wing. *Larch: Languages and Tools for Formal Specification*. Springer-Verlag, 1993.

[66] Henri Habrias and Marc Frappier, editors. *Software Specification Methods*. ISTE, 2006.

[67] Anthony Hall. Seven myths of formal methods. *IEEE Software*, pages 11–19, September 1990.

[68] David Harel. Statecharts: A visual formalism for complex systems. *Science of Computer Programming*, 8:231–274, 1987.

[69] David Harel and P. S. Thiagarajan. Message sequence charts. http://www.comp.nus.edu.sg/ thiagu/public_papers/surveymsc.pdf.

[70] Eric C. R. Hehner. Predicative programming part i. *Commun. ACM*, 27(2):134–143, 1984.

[71] Eric C. R. Hehner. *A Practical Theory of Programming*. Springer-Verlag, 1993.

[72] Constance L. Heitmeyer, Alan Bull, Carolyn Gasarch, and Bruce Labaw. SCR*: A toolset for specifying and analyzing requirements.

In *COMPASS'95: 10th Annual Conference on Computer Assurance*, pages 109–122, June 1995.

[73] Constance L. Heitmeyer, Ralph D. Jeffords, and Bruce G. Labaw. Automated consistency checking of requirements specifications. *ACM Transactions on Software Engineering and Methodology*, 5(3):231–261, July 1996.

[74] Constance L. Heitmeyer, J. Kirby, and Bruce G. Labaw. Tools for formal specification, verification, and validation of requirements. In *COMPASS'97: 12th Annual Conference on Computer Assurance*, June 1997.

[75] Constance L. Heitmeyer and J. McLean. Abstract requirements specifications: A new approach and its application. *IEEE Transactions on Software Engineering*, 9(5):580–589, September 1983.

[76] Holger Herbst and Thomas Myrach. A repository system for business rules. In Robert Meersman and Leo Mark, editors, *Database Application Semantics, Proceedings of the Sixth IFIP TC-2 Working Conference on Data Semantics (DS-6)*, pages 119–139. Chapman & Hall, 1996.

[77] S. D. Hester, D. L. Parnas, and D. F. Utter. Using documentation as a software design medium. *Bell Syst. Tech. J.*, 60(8):1941–1977, October 1981.

[78] C. A. R. Hoare. *Communicating Sequential Processes.* Prentice Hall, 1985.

[79] C. A. R. Hoare. Programs are predicates. In C. A. R. Hoare and J. C. Shepherdson, editors, *Mathematical Logic and Programming Languages*, pages 141–155. Prentice Hall, 1985.

[80] James K. Huggins and Charles Wallace. An abstract state machine primer. Technical Report CS-TR-02-04, Michigan Technological University, December 2002.

[81] Daniel Jackson. Nitpick: A checkable specification language. In *Workshop on Formal Methods in Software Practice*, 1996.

[82] Daniel Jackson. Automating first-order relational logic. In *8th ACM SIGSOFT International Symposium on Foundation of Software Engineering*, 2000.

[83] Daniel Jackson. Alloy: A lightweight object modelling notation. *ACM Transactions on Software Engineering and Methodology*, 11(2):256–290, April 2002.

[84] Michael Jackson. *Software Requirements & Specifications.* Addison Wesley, 1995.

[85] Michael Jackson. *Problem frames: analyzing and structuring software development problems.* Addison Wesley, 2001.

[86] Michael Jackson. Problem frames and software engineering. *Information and SOftware Technology*, 47(14):903–912, November 2005.

[87] I. Jacobson, M. Christerson, P. Jonsson, and G. Overgaard. *Object-Oriented Software Engineering: A Use Case Driven Approach.* Addison-Wesley, 1992.

[88] Kurt Jensen. *Coloured Petri nets: basic concepts, analysis methods and practical use: volume 1.* Springer-Verlag, London, UK, second edition, 1996.

[89] Xiaoping Jia. An approach to animating Z specifications. In *COMPSAC 95*, 1995.

[90] Xiaoping Jia. *A Tutorial of ZANS - A Z Animation System (Version 0.31).* School of Computer Science, Telecommunication, and Information Systems, DePaul University, 1998.

[91] Xiaoping Jia. *ZTC: A Type Checker for Z Notation - User's Guide (Version 2.2).* School of Computer Science, Telecommunication, and Information Systems, DePaul University, 2002.

[92] Cliff B. Jones. *Systematic Software Development Using VDM.* Prentice Hall, second edition, 1990.

[93] Stefan Kahrs, Donald Sannella, and Andrzej Tarlecki. The defintion of extended ML: a gentle introduction. *Theoretical Computer Science*, 173(2):445–484, 1997.

[94] John Arnold Kalman. *Automated Reasoning with OTTER.* Rinton Press, 2001.

[95] Ferhat Khendek and Gregor v. Bochmann. Formal specifications design, evolution and reuse. In *CASCON '93: Proceedings of the 1993 conference of the Centre for Advanced Studies on Collaborative research*, pages 184–193. IBM Press, 1993.

[96] R. Kowalski. The relation between logic programming and logic specification. In C. A. R. Hoare and J. C. Shepherdson, editors, *Mathematical Logic and Programming Languages*, pages 11–27. Prentice Hall, 1985.

[97] Heiko Krumm. Temporal logc. http://www4.cs.uni-dortmund.de/RVS/MA/hk/Kru00.pdf.

[98] D. Richard Kuhn. A technique for analyzing the effects of changes in formal specifications. *The Computer Journal*, 35(6), December 1992.

[99] D. Richard Kuhn, Ramaswamy Chandramouli, and Ricky W. Butler. Cost effective use of formal methods in verification and validation. *Foundations Verification and Validation Workshop*, October 2002.

[100] Dean Leffingwell and Don Widrig. *Managing Software Requirements: A Unified Approach.* Addison-Wesley, 2000.

[101] Lin Liu and Eric Yu. From requirements to architectural design vusing goals and scenarios. In *STRAW'01: First International Workshop from Software Requirements to Architectures*, Toronto, May 2001. IEEE Press.

[102] Shaoying Liu, A. Jeff Offutt, Chris Ho-Stuart, Yong Sun, and Mitsuru Ohba. Sofl: A formal engineering methodology for industrial applications. *IEEE Trans. Softw. Eng.*, 24(1):24–45, 1998.

[103] L. Logrippo, M. Faci, and M. Haj-Hussein. An introduction to LOTOS: Learn by examples. *Computer Networks and ISDN Systems*, 23(5), February 1992.

[104] K. C. Mander and F. A. C. Polack. Rigorous specification using structured systems anaysis and Z. *Information and Software Technology*, 37(5–6):285–291, 1995.

[105] James Martin and Carma McClure. *Structured Techniques for Computing.* Prentice Hall, 1985.

[106] Peter McBrien, Anne Helga Seltveit, and Benkt Wangler. Rule based specification of information systems. In *Conference on Information Systems and Management of Data*, pages 212–228, 1994.

[107] R. Milner. *Communication and Concurrency.* Prentice Hall, 1989.

[108] Rick Mugridge and Ward Cunningham. *Fit for Developing Software: Framework for Integrated Tests.* Prentice Hall, 2005.

[109] M. D. Myers. Qualitative research in information systems. *MIS Quarterly*, 21(2):241–242, June 1997.

[110] Bashar Nuseibeh. Weaving the software development process between requirements and architectures. In *STRAW'01: First International Workshop from Software Requirements to Architectures*, Toronto, May 2001. IEEE Press.

[111] Foundations of Software Engineering Microsoft Research. AsmL: The abstract state machine language. http://www-madlener.informatik.uni-kl.de/teaching/ss2004/fsvt/14.05.04.main.b.pdf, October 2002.

[112] Girish Keshav Palshikar. Applying formal specifications to real-world software development. *IEEE Software*, pages 89–97, November 2001.

[113] D. L. Parnas, G. Asmis, and J. Madey. Assessment of safety-critical software in nuclear power plants. *Nucl. Safety*, 32(2):189–198, April 1991.

[114] James L. Peterson. Petri nets. *ACM Computing Surveys*, 9(3), September 1977.

[115] Nico Plat, Jan van Katwijk, and Hans Toetenel. Application and benefits of formal methods in software development. *Software Engineering Journal*, 7(5):335–347, September 1992.

[116] Fiona Polack. SAZ: SSADM version 4 and Z. http://www-users.cs.york.ac.uk/ fiona/PUBS/Habrias.ps.

[117] Fiona Polack. SAZ invoicing case study: Questions raised in development. http://www-users.cs.york.ac.uk/ fiona/PUBS/NantesInvoicing.ps, May 1997.

[118] Wishnu Prasetya. JUnit 4.x quick tutorial. http://code.google.com/p/t2framework/wiki/JUnitQuickTutorial, January 2009.

[119] Roger S. Pressman. *Software Engineering: A Practitioner's Approach.* McGraw-Hill, european adaptation fifth edition, 2000.

[120] Edward M. Reingold and Wilfred J. Hansen. *Data Structures.* Little, Brown and Company, 1983.

[121] Ronald G. Ross. *Principles of the Business Rule Approach.* Addison-Wesley, 2003.

[122] G. Rossman and S. F. Rallis. *Learning in the Field: An Introduction to Qualitative Research.* SAGE Publications, 2003.

[123] Gordon Rugg and Marian Petre. *A Gentle Guide to Research Methods.* Open University Press, 2007.

[124] James R. Rumbaugh, Michael R. Blaha, William Lorenson, and Frederick Eddy. *Object-Oriented Modelling and Design.* Prentice Hall, 1991.

[125] Joachim Schmid. Introduction to AsmGofer. http://www.tydo.de/AsmGofer/files/AsmGoferIntro.pdf, March 2001.

[126] Geri Schneider and Jason P. Winters. *Applying Use Cases - A Practical Guide.* Addison-Wesley, 1998.

[127] Rolf Schwitter. English as a formal specification language. In *Thirteenth International Workshop on Database and Expert Systems Applications*, pages 228–232, Aix-en-Provence, France, September 2002.

[128] Rolf Schwitter and Norbert E. Fuchs. Attempto - from specifications in controlled natural language towards executable specifications. In *EMISA Workshop Natrlichsprachlicher Entwurf von Informationssystemen*, Tutzing, Germany, May 1996.

[129] Rolf Schwitter, Anna Ljungberg, and David Hood. ECOLE23: A look-ahead editor for a controlled language. In *EAMTCLAW03, Controlled Language Translation*, pages 141–150, Dublin, Ireland, May 2003.

[130] C. T. L. L. Silva, J. F. B. Castro, and J. Mylopoulous. Detailing architectural design in the tropos methodology. In *STRAW'03: Second International Workshop from Software Requirements to Architectures*, Portland, Oregon, May 2003. IEEE Press.

[131] Ian Sommerville. *Software Engineering.* Addison-Wesley, fifth edition, 1996.

[132] Ian Sommerville. *Software Engineering.* Addison-Wesley, sixth edition, 2001.

[133] J. M. Spivey. *The Z Notation: A Reference Manual.* Prentice Hall, second edition, 1998.

[134] Robert F. Stärk, Joachim Schmid, and Egon Börger. *Java and the Java Virtual Machine: Definition, Verification, Validation.* Springer, 2001.

[135] Susan Stepney, Fiona Polack, and Ian Toyn. Refactoring in maintenance and development of Z specifications and proofs. In John Derrick, Eerke Boiten, Jim Woodcock, and Joakim von Wright, editors, *REFINE 2002, BCS-FACS Refinement Workshop, Copenhagen, Denmark, July 2002*, volume 70 of *ENTCS*, pages 396–415. Elsevier, 2002.

[136] Douglas A. Stuart, Monica Brockmeyer, Aloyius K. Mok, and Farnam Jahanian. Simulation-verification: Biting at the state explosion problem. *IEEE Trans. Softw. Eng.*, 27(7):599–617, 2001.

[137] D. Svetinovic. Architecture-level requirements specification. In *STRAW'03: Second International Workshop from Software Requirements to Architectures*, Portland, Oregon, May 2003. IEEE Press.

[138] A. Tsalgatidou, V. Karakostas, and P. Loucopoulos. Rule-based requirements specification and validation. In *CAiSE'90: Proceedings of the Conference on Advanced Information Systems Engineering*, pages 251–263, Stockholm, Sweden, May 1990.

[139] Kenneth J. Turner. Incremental requirements specification with LOTOS. *Requirements Engineering Journal*, 2:132–151, November 1997.

[140] A. van Lamsweerde. Formal specification: a roadmap. In A. Finkelstein, editor, *The Future of Software Engineering*, pages 147–159. ACM Press, 2000.

[141] A. van Lamsweerde, R. Darimont, and P. Massonet. Goal-directed elaboration of requirements for a meeting scheduler: Problems and lessons learnt. In *RE'95: IEEE International Symposium on Requirements Engineering*, pages 194–203, York, England, March 1995. IEEE Computer Society Press.

[142] A. J. van Schouwen, D. L. Parnas, and J. Madey. Documentation of requirements for computer systems. In *RE'93: IEEE International Symposium on Requirements Engineering*, pages 198–207, San Diego, California, 1993. IEEE Computer Society Press.

[143] David A. Watt. *Programming Language Syntax and Semantics.* Prentice Hall, 1991.

[144] Jim Woodcock and Jim Davies. *Using Z - Specification, Refinement, and Proof.* Prentice Hall, 1996.

[145] R. K. Yin. *Case Study Research, Design and Methods.* SAGE Publications, third edition, 2002.

[146] E. Yu. Strategic actor relationships modelling with *i**. A tutorial given at IRST/University of Trento, December 2001.

[147] Pamela Zave. An insider's evaluation of paisley. *IEEE Transactions on Software Engineering*, 17(3):212–225, March 1991.

[148] Jane Zhao and Klaus-Dieter Schewe. Using abstract state machines for distributed data warehouse design. In Sven Hartmann and John Roddick, editors, *APCCM2004: First Asia-Pacific Conference on Conceptual Modelling*, Dunedin, New Zealand, 2004. Australian Computer Society.

[149] M.K. Zimmerman, K. Lundqvist, and N. Leveson. Investigating the readability of state-based formal requirements specification languages. In *ICSE'02: 24th International Conference on Software Engineering*, pages 33–43, May 2002.

[150] James E. Zull. *The Art of Changing the Brain.* Stylus Publishing, 2002.