Preface

Objectives of this book

This book is an introduction to a set of software specification methods. Its targeted audience are readers who do not wish to read pages of definitions in order to understand the basics of a method. The *same case study* is used to introduce each method, following a rigorously uniform presentation format. Special care has been devoted to ensure that specifications do not deviate from the case study text. As much as the method allows, what is specified is what appears in the case study text. The benefits are twofold. First, the reader can easily switch from one method to another, using his knowledge of the case study as a leverage to understand a new method. Second, it becomes easier to compare methods, because the same behavior is specified in each case.

Each method presentation follows the same pattern. The concepts are progressively introduced when they are needed. To illustrate the specification process, questions that the specifier should raise during the analysis of the case study are stated. Answers are provided as if they were given by an imaginary client. The question/answer process guides the derivation of the specification. Interestingly, the questions raised depend on the method, which is illustrative of the differences between them. When a question is raised in one method and not in another, the reader has an issue to resolve: does the other method allow this question? If so, what should the answer be? As such, this book is a trigger to stimulate the reader's curiosity about specification methods; it does not intend to provide all the answers. More elaborate materials are referenced in each chapter for a deeper coverage.

Some definitions

A specification *method* is a sequence of activities leading to the development of a product, called a specification. A method should provide enough guidance on how to conduct the activities and on how to evaluate the quality of the final product. A *specification* is a precise description, written in some notation (language), of the client's requirements. A notation is said to be *formal* if it has a formal syntax and formal semantics. A notation is said to be *semi-formal* if it only has a formal syntax.

Several characteristics of a system can be specified. One may distinguish between

functional requirements, efficiency requirements and implementation requirements. Functional requirements address the input-output behavior of a system. Efficiency requirements address the execution time of a system. The client may be interested in specifying a time bound for obtaining a response from the system. Some authors argue[1] that a specification without time bounds is not an effective specification: indeed, strictly speaking, if the specification does not include a time bound, the implementation may take an arbitrary duration to provide a response. It is impossible to distinguish between an infinite loop and a program that takes an arbitrary time to respond. Implementation requirements address issues like the programming language to use, the software components to reuse, the targeted hardware platform, the operating systems. The methods described in this book address functional requirements.

Specifications as contracts

A specification constitutes a *contract* between the client and the specifier. As such, the client must be able to understand the specification, in order to validate it. Typically, clients are not sufficiently versed in specialized notations to understand a specification. There are several ways to circumvent this lack of familiarity. The least is to rephrase the specification in the client's natural language, avoiding ambiguities as much as possible. If the specification is executable, scenarios can be tested with the client. The use of examples and counter-examples is a good technique to ensure that the client and the specifier understand each other.

The specification is also a contract between the specifier and the implementor. Of course, it is expected that the implementor understands the notation used for the specification. The implementor may not be familiar at all with the client's requirements and his application domain. The natural langage description provided to the client is also essential to the implementor, because it justifies and explains the specification. It allows the implementor to map specification concepts to application domain concepts. The textual description is to a specification what explanations are to formulas in mathematics.

Risks of not using specifications

Developing a software system without a specification is a random process. The implementation is doomed to be modified, sometimes forever, because it never precisely matches the client's needs. The goal of a specification is to capture the client's requirements in a concise, clear, unambiguous manner to minimize the risks of failure in the development process. It is much cheaper to change a specification than to change an implementation.

Additionally, the specification must leave as much freedom as possible to the implementor, in order to find the best implementation in terms of development cost, effi-

[1] Hehner E.C.R. (1993) *A Practical Theory of Programming.* Springer-Verlag

ciency, usability and maintainability. Abstraction is a good mechanism to support implementation freedom. For instance, if a sort function must be specified, the specifier need not to specify that a particular sort algorithm should be used. The implementor is free to pick any sorting algorithm like quicksort or bubblesort. Non-determinism is another good mechanism to provide more freedom for the implementation. For instance, one may specify a function that changes a dollar for a set of coins by just stating that the sum of the coins is equal to one dollar. The specification need not to prescribe how the set of coins is selected. During the implementation, an algorithm that minimizes the number of coins may be used, or one that gets rid of five-cent coins first, in order to minimize the weight of the coins in the machine (just for the sake of the argument).

Even when the implementation is finished, the specification is very useful. Conducting maintenance without a specification is a risky, expensive business. To modify a program, one must first know what it does.

Validation of a specification

A fundamental issue is to make sure that the specification "matches" the client's needs. This activity is called *validation*. Note that we use the verb "match" instead of a stronger verb like "prove", or "demonstrate", in the definition of the validation concept. By its very nature, a specification cannot be "proved" to match the client's requirements. If such a proof existed, then it would require another description of the requirements. If such a description is available, then *it is* a specification.

A specification is the starting point of the development process. It has the same status as axioms of a mathematical theory. They are assumed to be right. Of course, one can prove that a specification is *consistent* (i.e., that it does not include a contradiction), just as one can prove that the axioms of a theory are consistent. But this is a different issue from validation.

Validation consists essentially of stating *properties* about the specification, and proving that the specification satisfies these properties. Properties describe usage scenarios at various levels of abstraction. They can refer to concrete sequences of events, or they can be general statements about the safety or the liveness of the system.

The more properties are stated, the more the confidence in the specification validity is increased. Properties are like theorems of a theory: they must follow from the specification. In summary, validation is an empirical process; a specification is deemed valid until one finds a desired property that is not satisfied.

Satisfaction of a specification

It must be possible to demonstrate that the implementation *satisfies* the specification. A first approach is to progressively *refine* the specification until an implementation is reached. If it is possible mathematically to prove that each refinement satisfies the specification, we say that the development process is *formal*. Another approach is to

test the implementation. *Test cases* are derived from the specification. The results obtained by running the implementation for these test cases are compared with the results prescribed by the specification. Such a development process is said to be *informal*. For most practical applications, it is not feasible to exhaustively test a system.

From a theoretical viewpoint, proving the correctness of an implementation is more appropriate than testing it. From a practical viewpoint, testing is easier to achieve. Since Gödel and Turing, we know the strengths and the limitations of formal development processes. For more than 30 years now, computer scientists have investigated the application of mathematics to the development of software systems, with the ultimate goal of developing techniques to prove that an implementation satisfies a specification. Progress has been made, but much remains to be done.

Tools

A semi-formal notation may be supported by tools like editors and syntax checkers. A formal notation, thanks to its formal semantics, may also be supported by interpreters, theorem provers, model checkers and test case generators. Support for informal notations is limited to general purpose editors using templates for documents.

Structure of the book

This book is divided in four parts. The first part includes state-based specification methods. In these methods, the description of the system behavior is centered around the notion of state transition. The operations (also called functions) of the system are specified by describing how their execution change the state of the system.

The second part is dedicated to event-based methods. An event is a message that is exchanged between the environment and the system. Event-based methods describe which events can occur and in what order. Some of these methods are related to state-based specifications, as they also describe state transitions. Others use process algebras or traces to describe the possible sequences of events.

The third part includes methods based on three quite different paradigms. The first method uses an algebraic approach. The system is described using sorts, operations and equations. Abstract data types are classical examples of algebraic specifications. The second method is based on higher-order logic and typed lambda calculus. Operations are defined as functions on the system state. The last two are based on Petri nets. A Petri net is a graph with two kinds of vertices: places and transitions. Tokens are assigned to places. The behavior of the system is represented by the movement of tokens between places using transitions.

To help the reader in understanding and comparing the methods, the last part provides a qualitative comparison of the methods based on a number of attributes such as paradigm, formality, provability, verification and graphical representation. It also includes a glossary of the most important concepts used in the chapters, providing a definition of their contextual usage.

Summary of changes in the second edition

The first edition of this book was published by Springer-Verlag London in 2001. This new edition welcomes six new chapters: four new methods (ASM, Event B, TLA+, and UML-Z), the comparison and the glossary. Finally, existing chapters have been revised to adjust their contents to reflect recent developments.

The case study

The next sections reproduce the text of the case study that was submitted to authors and the guidelines for preparing their specifications. The case study seems very simple the first time through. When reading the various solutions, one quickly finds that its detailed analysis is surprisingly stimulating.

The text of the case study

1. The subject is to invoice orders.
2. To invoice is to change the state of an order (to change it from the state "pending" to "invoiced").
3. On an order, we have one and one only reference to an ordered product of a certain quantity. The quantity can be different to other orders.
4. The same reference can be ordered on several different orders.
5. The state of the order will be changed into "invoiced" if the ordered quantity is either less or equal to the quantity which is in stock according to the reference of the ordered product.
6. You have to consider the two following cases:
   (a) Case 1
   All the ordered references are references in stock. The stock or the set of the orders may vary:

   - due to the entry of new orders or cancelled orders;
   - due to having a new entry of quantities of products in stock at the warehouse.

   However, we do not have to take these entries into account. This means that you will not receive two entry flows (orders, entries in stock). The stock and the set of orders are always given to you in a up-to-date state.
   (b) Case 2
   You do have to take into account the entries of:

   - new orders;
   - cancellations of orders;
   - entries of quantities in the stock.

The guidelines for preparing specifications

Perhaps you will consider that the case study text is incomplete or ambiguous. One goal of this exercise is to know what questions are raised by each method.

You may propose different solutions (expressing consistent requirements) and you will explain how your method(s) have brought you to propose these solutions.

The questions that you had to deal with in order to solve the case study should be stated according to the following guidelines:

- Questions must be on the problem domain. They are directed to the user. They must be specific.
- Questions are better answered by several answers (options); pick one answer to continue the analysis.
- Show what verifications your method has allowed you to do (e.g., detection of inconsistencies in the answers that you have chosen).

Finally, do not extend the domain. For example, do not specify stock management (e.g., when to restock, following what minimum quantity, etc.), do not add new information such as category of customer, category of product, payment modality, bank account, etc.

Warning

This book illustrates some specification methods using a single case study. Although it is an excellent approach, from a pedagogical viewpoint, to provide an overview and a basic comparison of methods, the reader should not conclude that it is sufficient to evaluate and select specification methods. Each method has it strengths and weaknesses. A single case study cannot claim to properly represent all of them.

Wishing to contribute?

We would like this project to continue to evolve. If you wish to solve this case study using your favorite method, please check the book's web page at:

`http://www.dmi.usherb.ca/~spec`

Your contribution and comments are welcome. The case study, guidelines, new solutions, comments about solutions and additional materials about specification methods will be available at this address.

Acknowledgements

This book is part of a long story. In 1994, Henri Habrias proposed to the community of software engineering the Invoicing case study. The first solution, with *SA/RT* and *SCCS*, was submitted by Andy Galloway (University of York, UK) and distributed to the participants of the École d'été *CEA-EDF-INRIA* in June 1995. Three years later,

an International Workshop on Comparing Systems Specification Techniques, titled *"What questions are prompted by one's particular method of specification?"* was co-organized in Nantes by M. Allemand, C. Attiogbé and H. Habrias in March 1998[2].

This book was developed and refined in a collaborative effort. Each contributor has reviewed chapters of other contributors. Their mutual suggestions and comments have significantly enriched the final version of this book. Andy Galloway and Steve Dunne kindly reviewed several chapters. Panawe Batanado provided precious help for typesetting of the final version, enjoying the intricate pleasure of LaTeX. We are grateful to all these people.

Marc Frappier
Henri Habrias
Sherbrooke and Nantes
March 2006

[2] M. Allemand, C. Attiogbé and H. Habrias, editors, (Invoice'98) International Workshop on *Comparing Systems Specification Techniques — What Questions are Prompted by One's Particular Method of Specification?*, Nantes, France, 26-27 March 1998, ISBN 2-906082-29-5.