A Pragmatic Approach to Formal Specification

Maritta Heisel
Technische Universität Berlin
FB Informatik – FG Softwaretechnik
Franklinstr. 28-29, Sekr. FR 5-6
D-10587 Berlin, Germany
heisel@cs.tu-berlin.de
fax: (+49-30) 314-73488

August 13, 1995

Abstract

I propose to overcome some difficulties arising in the practical usage of formal specification techniques by adopting a pragmatic attitude. I argue that the transition from informal requirements to a formal specification should not be made too early; that it is not necessary to formally specify every detail; that different formalisms should be combined where appropriate; and that sometimes it may be useful not to adhere to limitations imposed by the formal specification language.

1 Introduction

In theory, everybody knows the advantages of formal specifications:

- The problem is analyzed in more detail and thus better understood.
- The formal specification is an unambigous and (hopefully) complete starting point for the implementation of the system.
- The formal specification documents the behavior of the system.
- It can be used to choose test cases and to determine if the results of the test cases coincide which the expected behavior.
- It makes maintenance and evolution of the system easier.
- The product seems to contain fewer errors.

In practice, however, formal specification techniques are not widely applied for the following reasons:

1. The formal nature of the languages may make their use difficult, especially if the semantics is not easily understood by non-experts.
2. Formal specification languages can be as rich as programming languages and have a similar learning curve.
3. There is no single specification language that is equally well suited for all kinds of systems and all aspects of an individual system.

4. It takes longer and is more expensive to set up a formal specification than to specify a system with conventional methods.

Are these really valid arguments against formal specification, and if so, what can be done to ameliorate their effects? Point 4 cannot be regarded as a drawback because a greater effort in the earlier phases of software development pays off in later phases and does not lead to an overall increase of costs.

The difficulties mentioned in points 1 and 2 cannot be overcome completely. But programming languages are formal languages, too, and nobody argues against programming because this makes it necessary to learn one or more programming languages with a non-trivial semantics. What is needed are specification languages with useful, semantically clean and intuitively clear concepts. Such languages would make the introduction of formal specification techniques into software engineering practice much easier. Existing specification languages are not altogether bad, but all of them leave something to be desired, and unnecessarily so, as I have argued elsewhere [Hei95].

Even carefully designed languages have their strenghts and weaknesses. We cannot expect to find one single general-purpose specification language that in addition offers a manageable number of constructs. Hence, point 3 is a very serious one. A straightforward idea is to use several formalisms instead of one. When such a combination is done with care, a "hybrid" specification will be clearer, shorter and more comprehensible than a specification in only one language that is clumsy in parts because the language does not allow some relevant parts to be expressed elegantly and concisely. I would even go further and recommend not to use formal specification techniques at all for those aspects of a system that just cannot be specified formally in a satisfactory way. These aspects need not necessarily be nonfunctional, see Section 2.

In order to make formal specification techniques better applicable, it does not suffice to consider only those activities in software development that deal with formal objects. Before we can write down some formal text, we should have an idea of what we want to write down. This means that a detailed requirements elicitation is a very important prerequisite to make the application of formal techniques successful.

In Section 2, I explain in more detail what I understand as a pragmatic approach to formal specification, where I consider the situation where a new system is built. More often, however, it will be the case that some legacy code has to be used and maintained. Here, formal specification techniques can be of help, too, as I show in Section 3. Finally, I discuss what is gained by using the pragmatic approach.

2 Specifying New Systems

In the following, I list some activities that make up the approach. They should not be considered as isolated phases with no feedback between each other, but should be carried out partially in parallel and repeatedly, like in the spiral model of software engineering. A "later" activity can reveal errors or omissions in an "earlier" phase.

Phase 1 *Define all relevant notions of the application domain.*

It must be possible to talk about all relevant aspects of the system. For this purpose, all phenomena that might be of interest must be given names and be informally described as precisely as possible. Jackson and Zave [JZ95] call this a *designation set*. In the context of software architectures, one may define *criteria* that are relevant for a given architectural style, see [HKa].

Phase 2 *Define the requirements for the system to be built.*

The notions to defined in Phase 1 provide a *language* in which the requirements can be expressed. If some requirement or some phenomenon concerning the system cannot be expressed, then the application domain was not investigated carefully enough, and we have to go back to step 1. Conversely, in order not to forget some requirements, we should make sure that for each

of the relevant notions a statement for the new system is made (this may be the statement that some phenomenon will not be treated at all).

Phase 3 *Convert the requirements in a pragmatic way into a formal specification.*

By "pragmatic" I mean that we should not always adhere to the pure lore of formal specification but take the freedom to make specifier's life easier. In my past work on and with formal specification, I applied the following relaxations of formal specification discipline[1].

1. Combine different formal or semi-formal specification techniques if one formalism alone is not powerful enough to express all relevant parts of the specification elegantly.
2. If the specification would be as low-level as program code, refrain from specifying these details formally but use conventional specification techniques and document the code particularly detailed.
3. Ignore ristrictions of the specification language if it is clear how to express the requirement in a semantically sound way.

Of course, all the above relaxations must be applied with great care because they are potentially dangerous: a specification where several formalisms are combined may be inconsistent. Incomplete formal specifications may result in an insufficient understanding of the parts not formally specified and thus lead to a wrong implementation. Indeed, this relaxation should mostly be applied for legacy code, see Section 3. Finally, when we write down illegal expressions of a specification language, we should be sure that this still has a well-defined semantics, which should be explained carefully in the accompanying text of the formal specification. I now describe some examples and try to sketch how to apply the relaxations "safely".

Combining different formalisms. For the specification of a steamboiler [BHW],it is appropriate to specify the state transitions and the events triggering the transitions with Statecharts [Har90] and the details of the states with Z [Spi92]. It has to be shown that each state transition of the Statechart is "covered" by the Z specification, i.e. there is a Z operation relating the two states.

For the formal specification of security functionality classes [ITS91], the clearest and most abstract specification is algebraic. For the systems to be certified, however, Z is more appropriate. It must be shown that the Z specification is a correct "refinement" of the algebraic one. This can be established by working from both ends: doing algebraic refinements of the abstract specifications, and performing "abstractions" on the Z specification until the refined algebraic specification and the abstracted Z specification can be related by a one-to-one mapping of the involved constructs, see [HP].

Leave out details. In [HKb], the specification of an existing event-action system is given. For such systems, matching of events against event-action-specifications is very important. Matching of composite events can be defined in terms of matching of primitive events relatively easily. A specification of matching for primitive events, however, would be no more abstract than the code itself and would make the formal specification much longer and less comprehensible. We therefore decided not to include matching of primitive events in the formal specification but refer to the system documentation.

Ignore restrictions. For the specification of the Unix file system, [Hei95], I used a generic definition of trees where each node has a name, a content and an arbitrary number of successors. To define a function selecting a successor of a node with a given name, I did not want to give an executable specification but simply state that the function yield a node that is (i) a successor of the

[1] Other persons may come up with different relaxations, according to their experience.

given node and (ii) has the given name. A generic box in Z with this definition is illegal because in a generic box the predicates must uniquely define the declared items. There are several functions satisfying the specification, and I wanted to express that I don't care which one is implemented. I stuck this "illegal" specification because I was sure that it has a well-defined semantics.

Phase 4 *Set up a mapping between the requirements and the formal specification.*

Such a mapping shows where and how each requirement is reflected in the formal specification. It helps to make the specification complete. When the system must be adjusted to new requirements, the specification should be changed before the code. The mapping shows where the changes have to be made.

Phase 5 *Animate the specification.*

This step is very important for the validation of the specification. Usually, customers only have a vague idea of what the system should do. Having an executable prototype and being able to try things out helps a great deal in the elicitation of the "real" requirements.

Elsewhere [Hei95], I have argued that formal specifications should be as abstract as possible and that they should not introduce any implementation biases. Such specifications are usually not constructive and hence not executable. However, the possibility to animate the specification is so precious that I consider it worthwhile to perform a few refinement steps that make the specification executable.

Again, the different phases are not independent of each other. Especially phase 5 will have an effect on phases 1 and 2. After several rounds in the spiral consisting of the above phases, the specification should stabilize, with a high probability that the requirements are complete, the specification captures them adequately, and that customers and developers understand equally well what the system is supposed to do.

3 Dealing with Legacy Code

Building a new system entirely from scratch is more the exception than the rule. More often, legacy code has to be used and maintained. But also for existing code, it is very useful to have a formal specification. It documents the behavior of the system and helps in maintainance and evolution. More details can be found in [HKb].

To deal with legacy code, the approach described in the previous section has to be adjusted. Phase 1 does not change. In phase 2, not the reqirements are formulated but the behavior of the system as far as it is known. Phases 3 and 4 are as before. In phase 5, the formal specification is used to generate test cases. These test cases should be run in order to check if the results of phases 2 and 3 coincide with the actual behavior of the system.

Using reverse engineering tools, the formal specification can help in locating code that deals with a certain aspect of the given system. An example of this is presented in [HKa].

4 What is Gained by the Pragmatic Approach?

Performing phases 1 and 2 is a joint task for developers and domain experts. Thus, requirements can be formulated in terms both can understand. The relaxations recommended for phase 3 (if necessary), contribute to make the specification less complicated and better comprehensible. Phase 5 helps to make the specification complete and and conforming to the wishes of the domain experts.

When requirements change, the mapping between requirements and formal specification shows where changes have to be made in the formal specification. The corresponding code can be found with reverse engineering techniques, see Section 3.

References

[BHW] Robert Büssow, Maritta Heisel, and Matthias Weber. Specification of a steam boiler. manuscript, in preparation, 1995.

[Har90] David Harel. Statecharts: a isual formalism for complex systems. *Science of Computer Programming*, 8:231–274, 1990.

[Hei95] Maritta Heisel. Specification of the unix file system: A comparative case study. In V.S. Alagar and Maurice Nivat, editors, *Proc. 4th Int. Conference on Algebraic Methodology and Software Technology*, volume V36 of *LNCS*, pages 475–488. Springer-Verlag, 1995.

[HKa] Maritta Heisel and Balachander Krishnamurhty. Bi-directional approach to modeling architectures. Submitted for publication, 1995.

[HKb] Maritta Heisel and Balachander Krishnamurhty. Yeast – a formal specification case study in Z. Submitted for publication, 1995.

[HP] Maritta Heisel and Jan Peleska. joint work, in progress. 1995.

[ITS91] ITSEC. Information technology evaluation criteria. Commission of the European Union, 1991.

[JZ95] Michael Jackson and Pamela Zave. Deriving specifications from requirements: an example. In *Proceedings 17th Int. Conf. on Software Engineering, Seattle, USA*, pages 15–24. ACM Press, 1995.

[Spi92] J. M. Spivey. *The Z Notation – A Reference Manual.* Prentice Hall, 2nd edition, 1992.