Design Calculi in Software Development: Theory and Practice

Kees Middelburg*

United Nations University, International Institute for Software Technology, P.O. Box 3058, Macau; E-mail: cam@iist.unu.edu; Fax: +853 712 940

Design calculi, also known as formal methods, allow to express descriptions of software systems formally, i.e. in a mathematically precise way, and to calculate properties of single descriptions and relations between pairs of them. Theoretical computer science has provided the foundations of practically useful design calculi, such as RAISE (Rigorous Approach to Industrial Software Engineering) [1, 2]. It has been demonstrated that the software development process and the resulting product can be improved by using such design calculi. In particular, design calculi facilitate the development of reliable, adaptable and reusable software systems. Nevertheless, there is still much resistance to use them.

First of all, I will explain the nature of formal descriptions and the importance of the ability to calculate properties of them and relations between them. Thereafter I will discuss the existing resistance to use design calculi and a policy to take away this resistance, pursued at various places in Europe and at UNU/IIST.

A design calculus offers the possibility to start the actual development of a software system by creating a formal specification for it. A formal specification of a software system is a formal description that:

- arises before the software system is constructed by a suitable abstraction from an application;
- conveys all that we may legitimately expect from the software system to be constructed;
- serves as a frame of reference against which the correctness of the eventual software system can be established.

A design calculus provides means to calculate properties of the formal specification and thus to validate it, i.e. to check whether it meets the requirements for the system.

The result of subsequent design steps, also known as refinements, can be recorded in more concrete formal descriptions, containing more implementation details. For each design step, we are able to calculate relations between the old description and the new one and thus to verify the design step, i.e. to check

*On leave (1996–1997) from KPN Research, Dept. of Network & Service Control, and Utrecht University, Dept. of Philosophy, the Netherlands.

Postprint from EU-China High Tech Conference, pages 307–311, SSTCC and EU DG XII, 1996.

whether the properties represented by the old description are preserved in the new one. After a number of steps, a formal description arises that can be automatically implemented – whether it is efficient depends upon the design decisions made.

Summarizing, a typical sequence of stages in development using a design calculus is:

- specification;
- validation of the specification;
- design steps consisting of:
  - refinement of a previous description,
  - verification of the refinement;
- implementation of the final description.

It is easy to see that in this way we can ascertain, in an early stage of development and to a high degree of precision, that the software system to be developed will match the user's requirements; and that we can establish, while developing the software system, a high degree of certainty that it will satisfy its specification, i.e. that it will be reliable. Besides, all details relevant to the adaptation of the system and the re-use of parts of the system or their design are recorded in a fully precise way.

Why is there so much resistance to use design calculi? The informal, in particular graphically based, techniques used in existing practice, are intuitively comprehensible to their user community. This is much less the case with most existing design calculi. Besides, introducing design calculi amounts to revolutionizing industrial practice. My conclusion is that acceptance requires a smooth evolution from techniques used in existing practice towards full-fledged design calculi.

Let me give an example from my experience. In the telecommunications field, SDL (Specification and Description Language) [3, 4] is widely used for specification and design. The first version of SDL became a recommendation of the ITU (International Telecommunication Union) in 1976. Since it has been extended several times. It originated from an informal graphical description technique already commonly used in the telecommunications field at the time of the first computer controlled telephone switches. In the telecommunications field, SDL has survived description techniques that are more design calculus oriented, such as LOTOS [5], and it will presumably still be used for a long time. Fig. 1 gives a description, using SDL's graphical representation, of the controller of a simple telephone answering machine.

It is interesting to see what tools and techniques there exist to complement SDL. It turn out that at present tools are available for:

- syntax-directed editing, syntax checking, etc.;
- simulation and limited checking of properties, test case generation;
- code generation.

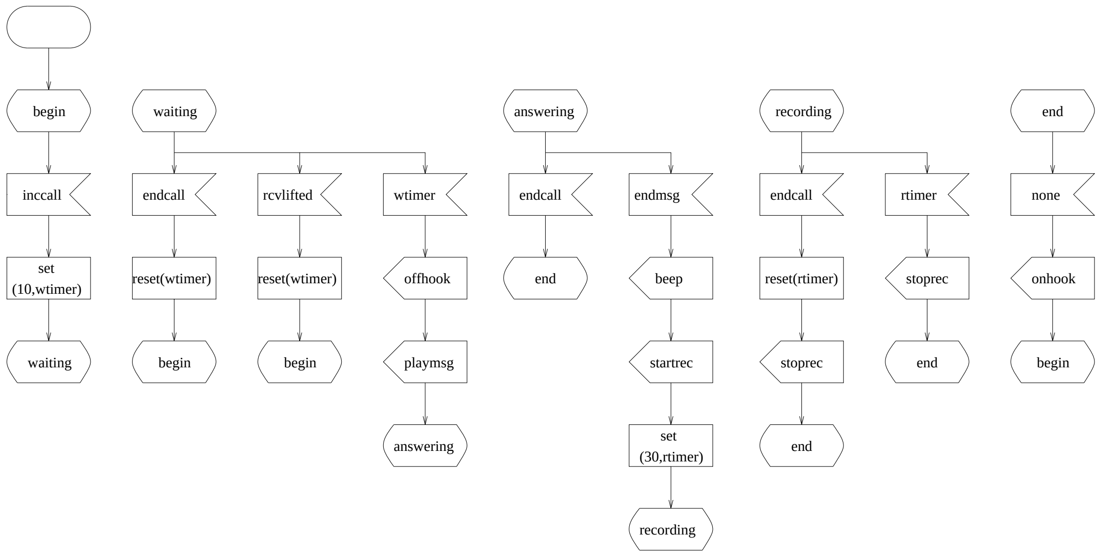

Figure 1: SDL description of a telephone answering machine

There is one complementing technique: Message Sequence Charts (MSCs), allowing very simple properties of SDL descriptions to be represented graphically. Given the available tools, it is not surprising that current practice in development using SDL differs from the one described above. A typical sequence of stages in development using SDL is:

- specification;
- limited validation of the specification;
- design steps consisting of refinement of a previous description;
- implementation of the final description;
- limited validation of the implementation.

To give an idea of what is meant here by limited validation, I will mention some properties of the telephone answering machine which should be respected by the behaviour described in Fig. 1, but which can not be checked by means of current tools:

- When the off-hook signal is issued to the network, nothing has happened since the detection of the last incoming call and meanwhile 10 time units have passed;
- When the recorder of the answering machine is stopped, at most 30 time units have passed since it was started.

In this way we can not ascertain to a reasonably high degree of precision that the software system will match the user's requirements; and we can only establish, after having developed the software system, a very low degree of certainty that it satisfies its specification.

This current practice is not in accordance with the needs in the development of telecommunications software using SDL. The intrinsic highly reactive and distributed nature of the systems developed in telecommunications demands more advanced analysis of SDL descriptions than currently possible. Besides, the increasing complexity is becoming a compelling reason to use, at least to a certain extent, formal verification to justify design steps. In other words, SDL should be complemented with techniques which would turn it into a full-fledged design calculus. However prerequisites for this are a dramatically simplified version of SDL and an adequate semantics for it. Only after that possibilities for advanced analysis can be elaborated and proof rules for formal verification devised.

Work in this area is, for example, being done at KPN Research and Utrecht University in the Netherlands and at UNU/IIST in Macau. At UNU/IIST we have the research programme DesCaRTeS (Design Calculi and Research for Telecommunications Systems) with the following primary aims:

1. to enhance the possibilities for advanced analysis of SDL descriptions, thus enabling better grounded validation of specifications.
2. to turn SDL into a full-fledged design calculus, thus enabling design steps made using SDL to be justified by formal verification.

To give an impression of what is involved here, I give the following list of research topics that are expected to be addressed:

- Operational semantics for a simplified version of SDL (SDL^-).
- Logics that are suitable to express the properties that can be represented by SDL^- descriptions.
- Tools that permit to check whether properties expressed in such logics are actually represented by given SDL^- descriptions.
- Semantic models for SDL^- that match the concepts around which SDL^- has been set up well.
- Semantics of SDL^-, based on such a model, that is fully abstract w.r.t. its operational semantics.
- Rules of reasoning for SDL^- which are sound with respect to its semantics.

Similar research projects, with respect to other techniques used in existing practice, are currently carried at a few places in Europe. I believe that this is an important development. Software engineering depends for its success on applying relevant computing science theory. For a long time, computing science was studying issues that were further and further ahead of the actual practical problems instead of, for example, trying to understand the concepts used in practice. The times are changing and here is an important area for academic and industrial research institutes in industrialized and developing countries to work together on the narrowing of gaps between theory and practice that are impeding software engineering to mature.

References

[1] RAISE Language Group. *The RAISE Specification Language.* Prentice-Hall, 1992.

[2] RAISE Method Group. *The RAISE Development Method.* Prentice-Hall, 1995.

[3] F. Belina, D. Hogrefe, and A. Sarma. *SDL with Applications from Protocol Specification.* Prentice-Hall, 1991.

[4] A. Olsen, O. Færgemand, B. Møller-Pedersen, R. Reed, and J.R.W. Smith. *Systems Engineering Using SDL-92.* Elsevier (North-Holland), 1994.

[5] LOTOS - a formal description technique based on the temporal ordering of observational behaviour. International Standard ISO 8807 (draft final text), 1988.