Formal Methods for the Quality Assurance of Digital Systems

Gerald Lüttgen*

Abstract

Formal Methods aim at improving the *reliability and safety of digital systems* by applying *mathematical specification and verification techniques*. This article gives an introduction to Formal Methods and illustrates them by means of an example dealing with parts of a *flight-guidance system*.

Introduction

Formal Methods is an area of research in Computer Science which aims at improving the *reliability* and *safety of digital systems* by applying *mathematical specification and verification techniques*. It complements traditional approaches to quality assurance such as reviews and testing.

The absence of malbehavior is of particular importance for *safety-critical systems*, such as used in traffic control, medicine, aeronautics, and aerospace. Experience shows that many fatal errors occur very rarely and under exceptional circumstances; one major source being the intrinsic interplay of *concurrent processes* as well as the inherent complexity of *fault-tolerant systems*. Unfortunately, even sophisticated testing methods do not catch some of these bugs, as is testified by many incidents, such as the floating point bug in Intel's Pentium processor, several airplane crashes, and the Ariane 5 blow-up. Hence, rigorous techniques are needed for guaranteeing the *correctness* of critical computer systems.

The philosophy underlying Formal Methods is similar to other scientific disciplines, such as Computational Fluid Dynamics. It relies on the *construction and mathematical analysis of models* in order to predict real-world systems' behavior. This approach often reveals potential weaknesses in a product's design, which may or may not have been discovered by standard engineering methods. Thereby, it also saves enormous costs associated with testing digital systems or experimenting in wind-tunnels. In addition, for safety-critical systems the analysis of the complete behavior of an abstracted model has been proved to produce better results than conducting tests which cover only part of a system's behavior. In contrast to systems studied in fluid dynamics the behavior of digital systems is not continuous. A slight change in the operating environment may cause abrupt changes in a system's behavior. Thus, the mathematics employed in Formal Methods is founded on *logic* and *deduction*, as opposed to differential equations and numerical calculation.

Formal Specification

The spectrum of *formal specification languages* for the modeling of systems ranges from *higher-order logics*, over *process-algebraic formalisms*, e.g. LOTOS, to *graphical languages*, such as Statecharts. Many specification languages consist of a small set of elementary constructs for which a rigorously *operational*, *denotational*, or *axiomatic semantics* is defined. In order to make them appealing to engineers, syntactic sugar, type systems, and graphical interfaces are often added. Some of the languages also provide the possibility to embed desired system properties within the specification, e.g. by including *assertions*.

Table 1: Example Specification in PVS

```
mode  : type = {cleared,active}
event : type = {switch,activate,deactivate,clear}
signal: type = {activated,deactivated,null}

react(m:mode, e:event) : [mode, signal] =
  if cleared?(m) then
    cond
      switch?(e)     -> (active, activated)
      activate?(e)   -> (active, activated)
      deactivate?(e) -> (m,      null)
      clear?(e)      -> (m,      null)
    endcond
  else
    cond
      switch?(e)     -> (cleared, deactivated)
      activate?(e)   -> (m,       null)
      deactivate?(e) -> (cleared, deactivated)
      clear?(e)      -> (cleared, null)
    endcond
  endif
```

Tables 1 & 2 give a flavor for formal verification languages. They depict the model of a simple component of a flight-guidance system, the process `react`, in the specification languages of the verification tools PVS [5] and

*Institute for Computer Applications in Science and Engineering (ICASE), Mail Stop 403, NASA Langley Research Center, Hampton, VA 23681-2199 (email: luettgen@icase.edu). This work was supported by the National Aeronautics and Space Administration under NASA Contract Nos. NAS1-97046 and NAS1-19480 while the author was in residence at ICASE.

SPIN [4], respectively. Process react continuously reacts to an external event by potentially changing its mode and sending a responding signal to its environment. Note that PVS has an expressive type system and adapts a functional style, although it is actually a higher-order logic, whereas SPIN has a simple type system and an imperative look and feel.

Table 2: Example Specification in SPIN

```
#define cleared 0
#define active  1

mtype  = {switch,activate,deactivate,clear,
          activated,deactivated,null       }
byte m = cleared;
chan e = [0] of {mtype};
chan s = [0] of {mtype};

proctype react
{do
 :: m == cleared ->
    if
    :: e?switch     -> m=active; s!activated
    :: e?activate   -> m=active; s!activated
    :: e?deactivate ->           s!null
    :: e?clear      ->           s!null
    fi
 :: else ->
    if
    :: e?switch     -> m=cleared; s!deactivated
    :: e?activate   ->            s!null
    :: e?deactivate -> m=cleared; s!deactivated
    :: e?clear      -> m=cleared; s!null
    fi
 od}
```

Formal Verification

Formal verification uses *mathematical proof* for showing the presence of desired and the absence of undesired system behavior. Two major research directions have been pursued: *theorem proving* [5] and *state exploration* [2].

In theorem proving systems and properties are expressed in a higher-order logic. A tool, called theorem prover, is then used for establishing that a system satisfies a property. The user interacts with the tool by invoking *basic prover commands* and *proof strategies*. The basic prover commands include *deduction rules*, *induction schemes*, and *decision procedures* for the built-in data types. Proof strategies combine basic proof steps together with *proof heuristics* in order to perform or find parts of proofs automatically. Although proof strategies continuously mature, theorem provers need to be guided by the user. They essentially assist the user by checking proof steps and by discharging simple proof obligations automatically. However, proof ideas have to be provided by the user, and proofs must be designed and constructed, having efficiency, reusability, and tool support in mind. Reconsider the flight-guidance system example, and assume that the system possesses two processes in the style of react with modes m1 and m2, respectively. Assume further that one wants to prove that whenever m1 is cleared then m2 is active. In PVS this property can be stated as

```
forall (p:paths) :  (ext(p)(m1)=cleared)
   implies (ext(p)(m2)=active)
```

where the type paths encodes the set of all system paths and the function ext(p) extracts the system state which is reached by following path p. The above property can then be proved by induction on the length of p.

An interesting fragment of digital systems, namely those whose behavior can be modeled by *finite-state machines*, such as *communication protocols* and *hardware*, is accessible to completely *automated verification methods*. Such methods are referred to as state-exploration methods as they are based on inspections of state machines. The most successful state-exploration method employed today is *model checking*. There, properties are specified in some *temporal logic* which can express system *safety* – e.g. deadlock-freedom – and *liveness* – e.g. a stable system state can eventually be reached. The property mentioned above can be stated in the temporal logic adapted in SPIN as follows:

```
[](m1==cleared -> m2==active)
```

where "[]" stands for "always", "->" for implication, and "==" for equality. Note that the operator [] avoids the complicated formulation of "reachable states" as is necessary with respect to the PVS model. Model checking algorithms, which are based on *fixpoint computation* or *automata-theoretic techniques*, can then determine whether a given state machine satisfies a given temporal formula. In case a model violates a property, model checkers return *diagnostic information* which allows the user to detect the exact cause of malfunctioning in the model under consideration.

Both theorem proving and state exploration have their strengths and weaknesses. Theorem proving is a general verification framework which is expressive enough to handle almost any verification task. The price one has to pay is the required user interaction. Consider for example the verification of a program loop. Once the user provides a suggestion for the *loop invariant*, a theorem prover can assist in checking whether it is indeed an invariant. In contrast, state exploration techniques automatically construct invariants. However, their application is usually restricted to finite-state systems, where for all practical purposes 'finite' means actually 'small'. For hardware verification an efficient data structure, namely *Binary Decision Diagrams* (BDDs), has been proposed which allows

for a compact representation of sets of states. Unfortunately, software systems are much more complicated to deal with. One approach is to combine theorem proving and state-exploration methods as follows. The real-world system is first abstracted in a way that the resulting model is a finite-state machine and that the abstraction preserves the properties of interest. The model is then amenable to state-exploration methods, and one can use theorem proving for proving that the properties of the model indeed carry over to the real system.

Formal Methods in Practice

In the past most research efforts in Formal Methods have been devoted to develop theoretical foundations and prototypic tools. Less attention has been paid to putting formal techniques into practice which requires to address the following questions: which phases of a *system's life-cycle* can profit most from Formal Methods, and what demands must formal *verification tools* meet?

Regarding the first question, it has been widely agreed upon that Formal Methods are particularly suitable for being employed in the *requirements phase*, i.e. early in the life-cycle. Whereas existing formal tools, such as type checkers or test suites, successfully catch errors that are introduced during the implementation phase, tools employed for requirements analyses have been proved to be less reliable. Errors made in the requirements phase often remain undetected until the delivery of the product to the customer. Problems with *faulty*, *incomplete*, or *inconsistent requirements* were experienced e.g. with software for the International Space Station [3]. Giving requirements not only a formal appearance but also a rigorous semantics makes them accessible to Formal Methods and is the key to circumvent the mentioned problems.

Concerning the second question, a successful verification tool has to *integrate* an attractive specification language together with intuitively to use validation and verification routines that do not suffer from cryptic input languages or unreadable outputs, as academic prototypes do. Moreover, verification tools should be highly automated and provide natural means for human guidance whenever necessary. Last but not least, they need to be satisfactory integrated with standard system-engineering methods.

Conclusions and Outlook

Experiences with Formal Methods emphasize their utility for analyzing digital systems. They do not compete with, but complement, existing approaches to quality assurance and help to increase confidence in the correctness of safety-critical systems. While the acceptance of Formal Methods is continuously growing, especially in the hardware and aeronautics industry, their transition from theory to practice has just started. On the one hand, the application of Formal Methods requires *training* and *experience*; on the other hand, it is unclear to what extent Formal Methods will be recognized by federal authorities for the *certification* of safety-critical systems. Remarkably, the corresponding FAA document does neither exclude nor require the application of Formal Methods [6].

ICASE's Formal Methods program aims at evaluating and developing verification techniques for aeronautic applications, such as *flight-guidance systems* and *integrated modular avionics*, thereby supporting NASA Langley's Formal Methods program [1].

References

[1] R.W. Butler et al. NASA Langley's research and technology transfer program in formal methods. In *COMPASS'95*, Gaithersburg, MD, 1995.

[2] E.M. Clarke et al. Automatic verification of finite-state concurrent systems using temporal logic specifications. *ACM Transactions on Programming Languages and Systems*, 8(2):244–263, 1986.

[3] S. Easterbook et al. Experiences using lightweight formal methods for requirements modeling. *IEEE Transactions on Software Engineering*, 24(1), 1998.

[4] G. Holzmann. *Design and Validation of Computer Protocols*. Prentice-Hall, 1991.

[5] S. Owre et al. Formal verification for fault-tolerant systems: Prolegomena to the design of PVS. *IEEE Transactions on Software Engineering*, 21(2):107–125, 1995.

[6] J. Rushby. Formal methods and digital systems validation for airborne systems. NASA Contractor Report 4551, 1993.

About the Author

Gerald Lüttgen received a Diploma in Computer Science from RWTH Aachen, Germany, in July 1994 and a Doctoral Degree in Computer Science from the University of Passau, Germany, in May 1998. His technical advisor was Prof. Rance Cleaveland at North Carolina State University, where he has been a visiting researcher in 1995/96, funded by a doctoral grant from the German Academic Exchange Service. He joined ICASE in October 1998 as a Staff Scientist. His research interests are in the formal specification, analysis, and verification of concurrent and distributed systems.