Benefits and Problems of Formal Methods

Martin Gogolla

University of Bremen, Computer Science Department
Database Systems Group, D-28334 Bremen, Germany
gogolla@informatik.uni-bremen.de

Abstract. Formal methods for software development have been discussed for decades. This paper will try to explain when and under what circumstances formal methods and languages in general and the Object Constraint Language (OCL) in particular can be employed in a beneficial way. The success of using a formal approach is highly influenced by the expectations and pre-requisite knowledge of the developers, the role the formal support in the development process is given, and of course by the used tools.

1 Introduction

Formal methods (FMs) for software development have been discussed for decades. The enormous spectrum ranges from naive mathematical set theory over various approaches for model or property oriented specification to newer developments like the Object Constraint Language (OCL) being an integral part of the Unified Modeling Language (UML). People's opinions on formal methods are naturally determined by their experience and knowledge on the subject. In the literature, many positive and also negative views on FMs have been expressed [Hal90,BH95a,BH95b,Gla96,HB96,KDGN97,Sch00].

The aim of this paper is (A) to give a brief introduction and overview on the terminology of formal methods, (B) to shortly debate the various statement papers on formal methods in the light of benefits and problems, and (C) to discuss in this context our own formal method, namely the support the UML Specification Environment USE [RG00] provides for the OCL [WK03].

The structure of the rest of this paper is as follows. Section 1 recites Bjorner's view of important concepts within formal methods. Section 2 repeats the central statement of various position papers on FMs and gives a structuring for the statements collected. Section 3 sets some of those statements into the context of our own work on OCL. Section 4 finishes the paper with a few concluding remarks.

2 Terminology

For central notions relevant within the software development process in the context of formal methods, we follow the definitions and explanations by Bjorner [Bjo98].

Method: A set of principles for selecting and applying techniques and tools in order effectively to analyze and synthesize an effective artifact, here software.
Formal method (FM): A set of principles for formal specification and design calculi techniques.
Design calculi: A set of proof or specification transformation rules.
Formal specification: Describes a model in a comprehensive and consistent way. It is expressed in a formal language. A model can be given, e.g., for an application domain, for a requirement, for a software architecture, or for a program organization.
Formal language: Specifies at meta-level (A) a syntax, (B) a (possibly model-oriented, e.g., denotational or structural operational) semantics, and (C) a proof system.
Tools for design calculi: Aspects (A) and (B) allow for tools that support input, editing, syntactic and abstract interpretation processing of specifications. Aspect (C) allows for tools that support reasoning (automated theorem proving, mechanically assisted proofs) about specifications.
Formal development: All specifications are formal. All proof obligations have been identified and formally proved using the proof system.
Rigorous development: All specifications have been expressed in the formal specification language and relevant theorems have been formally expressed, but the reasoning processes have referred to model-theoretic properties of the modeled domain, requirements or software.
Systematic development: All specifications have been expressed in the formal specification language and reasonable relations between development steps have been informally reasoned.

3 Myths, Myths and Other Beliefs

Properties of and opinions on formal methods (FMs) have been discussed in the literature for quite a while. In this section we shortly recite and try to structure the work done in:

- “Seven Myths of Formal Methods” [Hal90]
- “Seven More Myths of Formal Methods” [BH95a]
- “Ten Commandments of Formal Methods” [BH95b]
- “Formal Methods are a Surrogate for a More Serious Software Concern” [Gla96]

– “Impediments to Industrial Use of Formal Methods” [HB96]
– “Why Are Formal Methods Not Used More Widely?” [KDGN97]
– “Why Doesn’t Anyone Use Formal Methods” [Sch00]

In [Hal90] seven so-called myths of FMs are expressed, and this list is extended by another seven myths in [BH95a]. The claim of these ‘Myths’ papers is that these statements about FMs are often expressed but that these statements do not hold. The ‘Ten Commandments’ paper [BH95b] is written to help in the application of FMs in an industrial setting, and the intended audience includes both developers and users of FMs. The ‘Beliefs’ paper and the ‘Impediments’ papers both look at FMs from a more critical perspective and frankly point to weak points of FMs. The ‘Criteria’ paper [KDGN97] mainly looks at FMs from the software development point of view and discusses the relationship between FMs and the software development cycle and its phases. The ‘Aspects’ paper [Sch00] states requirements for what FMs should be able to do and which properties they should have.

The tables in Fig. 1 and Fig. 2 repeat the central statements of the mentioned papers. The reader is invited to go through these statements. But be warned: Some of the short statements have been distilled by the author of this paper (in particular for [Gla96,HB96]) and might not find acceptance by the original authors.

All these papers express certain statements relevant in the context of FMs. We have classified these statements into seven topics: General criticism, Development, Understanding, Tools, Properties, People, Application domain. Independent of whether the paper is inclined positively and negatively towards FMs the addressed statements represent central points about FMs. The tables in Fig. 1 and Fig. 2 show our statement classification and the corresponding statements from the respective papers. Interestingly, a number of statement topics is addressed in more than one paper. Let us shortly motivate and explain our classification (in the reverse order as opposed to the tables).

Application domain: Some statements concern the application domain or the applicability for FMs. Historically, FMs have many origins in the area of proving program correctness and were used in safety-critical applications, but they might be used in a different scope (not merely for program proving) and in all other application fields.

People: A second class of statements deals with people involved in the development process. It seems to be important that for successful use of FMs developers are well-trained and experts are available.

Properties: Another group of statements expresses properties with respect to computer science principles. FMs seem to have to follow the principles of imperative programming and object-oriented development.

Tools: FMs must be accompanied by tools just as programming languages are supported by compilers. However, the tools must be embedded well into the

General criticism	Belief-1: FMs are underdefined [Gla96]
	Belief-2: FMs are underevaluated [Gla96]
	Belief-3: FMs inhibit software malleability [Gla96]
	Belief-5: FMs will not lead to any breakthrough in software productivity/quality [Gla96]
	Impediment-2: Researchers worked on toy examples and industrially irrelevant problems [HB96]
	More Myth-5: FMs are not required [BH95a]
Development - General	Commandment-02: Thou shalt formalize but not over-formalize [BH95b]
	Commandment-08: Thou shalt not be dogmatic [BH95b]
	Criterion-01: Coverage: Describe all of an application or operate with other notations [KDGN97]
	Criterion-02: Integration with other components [KDGN97]
	Criterion-03: Group development: Combine efforts of several people [KDGN97]
	More Myth-7: FMs people always use FMs [BH95a]
Development - Other Methods	Commandment-05: Thou shalt not abandon thy traditional development methods [BH95b]
	More Myth-3: FMs mean forsaking traditional engineering design methods [BH95a]
Development - Cost	Commandment-03: Thou shalt estimate costs [BH95b]
	More Myth-1: FMs delay the development process [BH95a]
	Myth-5: FMs increase the cost of software development [Hal90]
Development - Evolution	Aspect-4: Specifications must live [Sch00]
	Criterion-04: Evolution of specification and ease its change [KDGN97]
Development - Quality	Commandment-07: Thou shalt not compromise thy quality standards [BH95b]
	Commandment-09: Thou shalt test, test, and test again [BH95b]
Development - Design	Criterion-06: Compatibility with design tools [KDGN97]
	Criterion-07: Compatible with design methods [KDGN97]
	Criterion-08: Communicate desired system characteristics to designers [KDGN97]
	Criterion-09: Facilitate design process [KDGN97]
Development - Implementation	Criterion-10: Implementation performance [KDGN97]
	Criterion-11: Support for unit testing in implementation phase [KDGN97]
Development - Maintenance	Criterion-12: Maintenance comprehension [KDGN97]
	Criterion-13: Maintenance changes [KDGN97]

Fig. 1. Statements - General criticism, Development

Understanding	Aspect-1: Specifications should be easy to understand [Sch00]
	Belief-4: FMs are not readable [Gla96]
	Commandment-01: Thou shalt choose an appropriate notation [BH95b]
	Commandment-06: Thou shalt document sufficiently [BH95b]
	Criterion-05: Usability: Annotate specifications [KDGN97]
	Myth-6: FMs are unacceptable to users [Hal90]
Tools	Aspect-5: Specifications must be supported by tools [Sch00]
	Aspect-6: Specification tools must be compatible and interoperable [Sch00]
	Aspect-7: Specifications must be easily portable and available [Sch00]
	Impediment-1: Existing tools are not production-quality [HB96]
	Impediment-3: Build-it-and-they-will-come - Research is often ignorant of industry challenges [HB96]
	More Myth-2: FMs are not supported by tools [BH95a]
	More Myth-6: FMs are not supported [BH95a]
Properties	Aspect-2: Specifications should be state-oriented [Sch00]
	Aspect-3: Specifications should support abstraction, subtyping and modularization [Sch00]
	Commandment-10: Thou shalt reuse [BH95b]
People	Aspect-8: Specification building must be taught [Sch00]
	Commandment-04: Thou shalt have a FMs guru on call [BH95b]
	Myth-4: FMs require highly trained mathematicians [Hal90]
Application domain	More Myth-4: FMs only apply to software [BH95a]
	Myth-1: FMs can guarantee that software is perfect [Hal90]
	Myth-2: FMs are all about program proving [Hal90]
	Myth-3: FMs are only useful for safety-critical systems [Hal90]
	Myth-7: FMs are not used on real, large-scale software [Hal90]

Fig. 2. Statements - Understanding, Tools, Properties, People, Application domain

development process and offer useful features for everyday practical work. It is not important support fancy theoretical frameworks.

Understanding: A major topic seems to be the understanding (or perhaps better the understandability) of FMs. A clear notation and means for documentation are expressed as fundamental.

Development: We have classified a huge group of statements under the topic 'Development' and have structured it again into: General remarks, Relationship to other (informal or semi-formal) methods, Cost, Evolution, Quality, and Development phases.

General criticism: The last, most critical but also most unspecific group of statements concerns general criticism. In our view, the most serious statement in this group is the one about software malleability. However, this may be just the other side of the possibility of enhanced quality through FMs.

We do not claim that the given classification is the only one or that the assignment of statements to topics is mandatory. The purpose of the tables is to stimulate discussion. In fact, one can view at the tables from inside or outside of FMs: In the inside view, an FM tool developer can use the table topics as a checklist and ask herself what relationships exist between her tool and the respective topic; in the outside view, a possible FM user can ask the methodologist or the tool developer about the connection between each topic and the method or tool.

4 OCL versus Statements on Formal Methods

OCL [OMG03,WK03] as part of the standardized UML has been developed as a language especially suited for object-oriented development. It has a Java- resp. C++-like syntax and claims to be easily applicable by regular developers. On the one hand, OCL is scientifically well-studied (see, e.g., the proceedings of the UML conference or the papers on the OCL workshops or [RG01,GR01] among many other interesting papers), on the other hand commercial tutorials and seminars are available for OCL. Commercial as well as university tools are around [TRF03]. OCL has been used extensively in the development of the UML 1.X and UML 2.X metamodels for expressing structural restrictions.

The basic idea of OCL is that merely diagrams as given by the graphical part of UML are not enough for a precise system description. Textual additions and tightenings in a formal language are necessary for avoiding ambiguities. OCL has emerged from practical needs to sharpen and tighten UML class diagrams, i.e., to formulate structural restrictions. Later OCL was also employed for pre- and postconditions of operations and in other UML diagrams, for example for guards in statechart diagrams.

However, OCL is not a classical formal specification language. Thus OCL does not have the burden of a classical formal method, but OCL as it stands today does also not have the rich background of a classical formal method (and a very high degree of acceptance within the classical formal methods community).

In the rest of this section we will discuss how some selected FM statements relate to OCL and in particular how they relate to our UML/OCL animation tool UML Specification Environment USE [RG00].

4.1 *Thou shalt choose an appropriate notation*

OCL expressions usually rely on a given class diagram formulated within UML. Thus the syntactic context, i.e., the classes, their attributes, operations, and associations, for OCL invariants and pre- and postconditions is determined graphically in an easy understandable way.

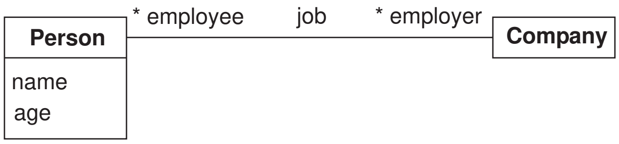

Fig. 3. UML Class Diagram

OCL now allows a short, condensed notation offering good possibilities for documentation of constraints. Let us consider the following invariant requiring that employed people are at least 18 years old which needs the class diagram sketched in Fig. 3.

```
context Person inv asEmployeeAtLeast18:
  employer->notEmpty implies age>=18
```

The mathematical content of the invariant could be stated as follows, but be aware that this mathematical formulation somewhat hides the context, the class Person, and does not mention the constraint name.

$(\forall\, p : Person \ ) \ (\ employer(p) \neq \emptyset \Rightarrow age(p) \geq 18 \ )$

Above we mentioned 'condensed notation' and 'documentation'. By condensed notation we mean that OCL knows implicit variables and implicit quantifiers and by documentation we refer to the possibility of specifying constraint names. Thus in the above invariant there an implicit universally quantified variable `self:Person` and, for example, the expression `age` is equivalent to `self.age`. In fact, the complete invariant is also equivalent to the following more verbose OCL expression.

```
context Company inv asEmployeeAtLeast18:
  Person.allInstances->forAll(self:Person |
    self.employer->notEmpty implies self.age>=18)
```

The body of the constraint explicitly introduces a universal quantification with a corresponding variable. This body expression is independent of its context, i.e., it does not use an implicit reference to `self`. Therefore in this case any context may be used, even the class Company which is not referenced with the constraint body. The usefulness of such shortcuts become important and apparent if one has longer constraints and not short ones like the above one. Be warned: One may be tended to underestimate the simple OCL feature to specify invariant names; this feature is a very useful, although trivial OCL feature; if you have OCL descriptions with, say 50 invariants, and you animate the specification short reference to satisfied and violated invariants is essential.

Another useful OCL feature is flattening of nested collections. For example, if you have an employee `ada` and you want to have the names of `ada`'s colleagues, i.e., the name of the employees which work in a company for which `ada` works, you simple write `ada.employer.employee.name` which has the type `Bag(String)`. Note that the type of the subexpression `ada.employer.employee` is flattened from `Bag(Set(Person))` to `Bag(Person)`.

4.2 *Specifications should support abstraction*

FMs support abstraction because they allow to ignore irrelevant implementation details. Such abstract descriptions often allow you to formulate your problem in a descriptive way by expressing only the desired properties, not yet an operational solution. However, with regard to executability the operational solution often has more attractive properties than the abstract, descriptive solution.

Consider the following example for describing the root function on the real numbers. The first solution, `rootSpec`, formulates the function in OCL using pre- and postconditions. The central requirement is that the difference between the number from which the root is to be calculated and the returned result multiplied with itself is below a given `precision`.

```
rootSpec(arg:Real,precision:Real):Real
context Root::rootSpec(arg:Real,precision:Real):Real
  pre  paramsOk: 0<arg and 0<precision and precision<=1
  post resultOk: (arg-result*result).abs<=precision and result>0
```

The root function may also be described in OCL in an operational way as a function returning a value. Note however, that the following description for the OCL function `rootImpl` is just one under many other implementations which satisfy the above specification with pre- and postconditions.

```
rootImpl(arg:Real,precision:Real):Real=
  rootImplAux(arg, precision,
              if arg>=1 then 1.0 else arg endif,
```

```
              if arg>=1 then arg else 1.0 endif)
rootImplAux(arg:Real,precision:Real,low:Real,high:Real):Real=
  let new:Real=(low+high)/2 in
  if (arg-new*new).abs<=precision
    then new
    else rootImplAux(arg,precision,
                     if new*new<arg then new else low endif,
                     if new*new<arg then high else new endif)
  endif
```

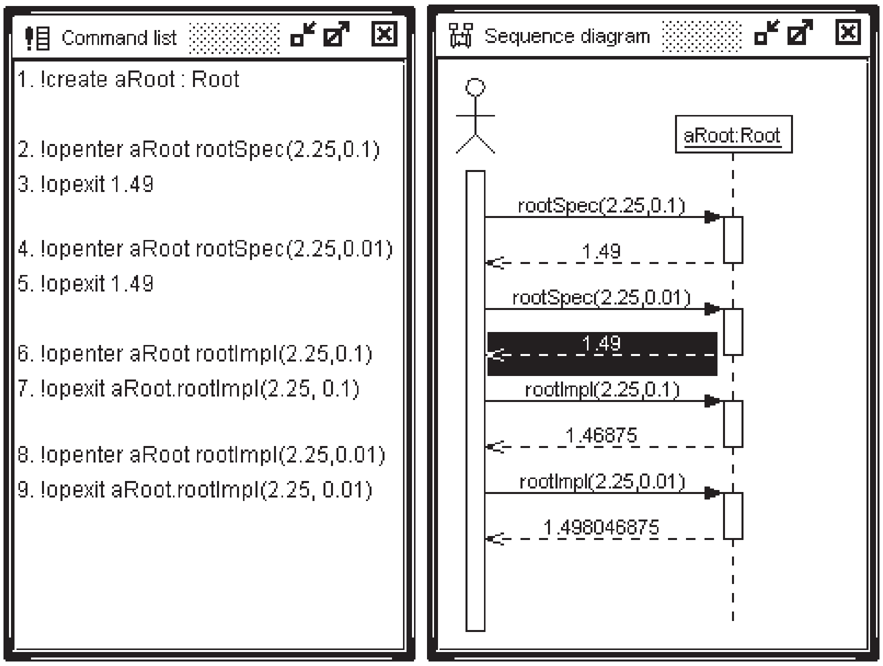

Fig. 4. Executing RootSpec and RootImpl

In Fig. 4 we have displayed part of a USE animation of the two functions in form of a UML sequence diagram. Even the descriptive specification with pre- and postcondition may be animated by providing explicit return values which are checked against the postcondition. The USE animation does use colors to draw special attention to particular points. The black-and-white inverted return arrow is displayed in USE in red. This indicates that the postcondition is not satisfied. In this case the provided return value `1.49` is not close enough to the correct result `1.5` under the given precision `0.01`.

4.3 *Specifications must be supported by tools*

Let us think of a simple traffic light application where one controller is responsible for four traffic lights in north, south, west, and east directions on a street crossing, and each traffic light is operated by one controller. Consider the UML class diagram for the controller and the traffic lights in Fig. 5 and the corresponding object diagram in Fig. 6. Does the object diagram meet the class diagram?

Fig. 5. UML Class Diagram for Traffic Light

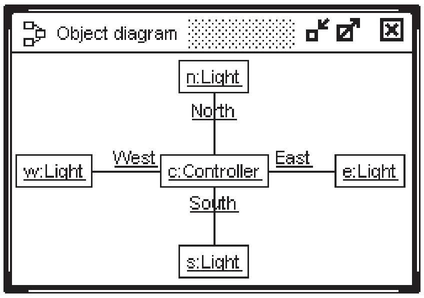

Fig. 6. UML Object Diagram

The multiplicities in the class diagram are not consistent with the object diagram! Consider the traffic light n. The class diagram requires that each traffic light is connected by the association North to one controller, which is true for the traffic light n. But it also requires this for the association South, West, and East. And this is not fulfilled for the traffic light n. In fact, this error can be found in one of my own papers. I did not check the 'obviously' true statement that the object diagram is consistent with the class diagram with our USE tool. The multiplicities are too restrictive. On the controller side they have to be changed to `0..1`.

In fact, one of main advantages and tasks of formal methods is that tools can reveal such design errors early in the development. Tools allow you to have on the one hand a general frame of your application world, in this case the class diagram with the multiplicities, and on the other hand concrete scenarios of your application world, in this case the object diagram. And the tools must check the consistency between the two.

4.4 *Specifications must live*

Continuing the traffic light example, if one changes the multiplicities on the controller side to `0..1`, it becomes allowed that traffic lights can exist which are not connected to any controller. Therefore, one should demand that for each traffic light exactly one of the four associations has to be established. Now OCL offers many alternatives to formulate this requirement.

```
context Light inv oneController_A:
  (controllerForNorth->size)+(controllerForSouth->size)+
  (controllerForWest ->size)+(controllerForEast ->size) = 1

context Light inv oneController_B:
  Bag{controllerForNorth,controllerForSouth,
      controllerForWest, controllerForEast}->size = 1

context Light inv oneController_C:
  Bag{controllerForNorth,controllerForSouth,
      controllerForWest, controllerForEast}->
  select(c|c.isDefined())->size = 1

context Light inv oneController_D:
  (controllerForNorth->size=1) xor (controllerForSouth->size=1) xor
  (controllerForWest ->size=1) xor (controllerForEast ->size=1)

context Light inv oneController_E:
  ( (controllerForNorth->size=1) and (controllerForSouth->size=0) and
    (controllerForWest ->size=0) and (controllerForEast ->size=0) ) or
  ( (controllerForNorth->size=0) and (controllerForSouth->size=1) and
    (controllerForWest ->size=0) and (controllerForEast ->size=0) ) or
  ( (controllerForNorth->size=0) and (controllerForSouth->size=0) and
    (controllerForWest ->size=1) and (controllerForEast ->size=0) ) or
  ( (controllerForNorth->size=0) and (controllerForSouth->size=0) and
    (controllerForWest ->size=0) and (controllerForEast ->size=1) )
```

Apart from the above five formulations, one can build further variations: Alternatives B' and C' with `Set` instead of `Bag` in B and C; alternatives D' and E' with `.isDefined()` instead of `->size=1` and `.isUndefined()` instead of `->size=0` in D and E. Thus we end up with nine possibilities. Which of these are equivalent? Which represent the semantics we want to express?

Formal methods need and require tools. These tools have to be based on the formal semantics of the formal specification language. But such tools are also highly needed in order to handle the inherent complexity of formal specification languages in order to understand the language constructs. One has to play around with alternatives, one has to see and feel how different formulations behave different: Specifications must live.

4.5 *Thou shalt test, test, and test again*

Let us continue the traffic light example by extending the class `Light` with a display attribute `signal:String` and an operation `showsGreen():Boolean = signal='G' or signal='YG'`. Furthermore three constraints restrict the possible snapshots or object diagrams.

```
context Light inv R_G_YG:
  signal='R' or signal='G' or signal='YG'
context Controller inv northSouth_WestEast_Coincide:
  north.signal=south.signal and west.signal=east.signal
context Controller inv northWest_Different:
  north.signal<>west.signal
```

The aim of the present considerations is to check whether, under the above assumptions, the danger of a crash is present, i.e., whether it is possible to construct a snapshot satisfying the following invariant.

```
context Controller inv crashPossible:
  north.showsGreen() and west.showsGreen()
```

To this aim, we design a special purpose procedure `generate` constructing snapshots by generating a controller and four traffic lights, by establishing the appropriate links, and by assigning attribute values.

```
procedure generate()
var theController:Controller, theLights:Sequence(Light);
begin
theController:=Create(Controller);
theLights:=CreateN(Light,[4]);
Insert(North,[theController],[theLights->at(1)]);
Insert(South,[theController],[theLights->at(2)]);
Insert(West,[theController],[theLights->at(3)]);
Insert(East,[theController],[theLights->at(4)]);
[theController.north].signal:=Try([Sequence{'R','G','YG'}]);
[theController.south].signal:=Try([Sequence{'R','G','YG'}]);
[theController.west].signal:=Try([Sequence{'R','G','YG'}]);
[theController.east].signal:=Try([Sequence{'R','G','YG'}]);
end;
```

This snapshot or test case generation procedure is embedded into the script shown below and is called after the invariant `crashPossible` has been dynamically added to the present invariants. Our test case generator responds with the sequence of commands that generate the snapshot where all invariants hold. In the concrete example we see that the given invariants do not guarantee that two adjacent traffic lights, here the north and west one, do not show green at the same time.

```
use> gen load traffic.invs
     Added invariants: Controller::crashPossible
use> gen start ... generate()
use> gen result
     Random number generator was initialized with 7686.
     Checked 45 snapshots.
     Result: Valid state found.
     Commands to produce the valid state:
     !create Controller1 : Controller
     !create Light1,Light2,Light3,Light4 : Light
     !insert (Controller1,Light1) into North
     !insert (Controller1,Light2) into South
     !insert (Controller1,Light3) into West
     !insert (Controller1,Light4) into East
     !set Light1.signal = 'G'
     !set Light2.signal = 'G'
     !set Light3.signal = 'YG'
     !set Light4.signal = 'YG'
```

Proceeding this way, formal methods support test case development. They may be used for test case design nicely because formal methods identify the distinguishing cases clearly as equivalence classes where certain formulas do hold or do not hold. Thus test cases can be generated in an automatic way.

This completes our in-depth discussion of the selected statements on formal methods. We have looked only at five typical statements from the comprehensive list, but we think that really all statements make a concern for formal methods in general and for our USE tool in particular.

5 Conclusion

In this paper we have recaptured central notions from formal methods, structured statements from papers estimating the value of formal methods and put some of these statements into the context of our USE tool.

The essence we have drawn from studying all these statement about formal methods is that the success of a formal method is not determined so much by its nice mathematical properties. We think the success is determined by the tools and in particular by the everyday usability of the tool. One aspect we see underdeveloped in formal method tools is the aspect of documentation possibilities (at least in our USE tool). And the tools and an underlying development method including a tight connection to other parts of the software development process must be documented and explained well. This helps that other people (not the developer of the formal method) can use the approach in a beneficial way. In our understanding, this is what the Build-it-an-they-will-come statement refers to.

On the other hand, working on the Build-it-an-they-will-come front and on the documentation front is scientifically a bit dangerous: If you submit a paper on many small improvements of the previous version of your tool, referees usually underestimate the contribution of this work and give preference to newer, fancier ideas which however are 'not yet' supported by a tool. So working in this direction with a pure research background must be considered well.

Acknowledgments

Thanks to Alfred Strohmeier for initiating the work on this papers. Arne Lindow has provided fruitful comments on draft versions of this paper.

References

[BH95a] Jonathan P. Bowen and Michael G. Hinchey. Seven More Myths of Formal Methods. *IEEE Software*, 12(4):34–41, July 1995.

[BH95b] Jonathan P. Bowen and Michael G. Hinchey. Ten Commandments of Formal Methods. *IEEE Computer*, 28(4):56–63, April 1995.

[Bjo98] Dines Bjorner. Formal Methods in the Twentyfirst Century: An Assessment of Today - Predictions for the Future. In *Proc. Int. Conf. Software Engineering (ICSE'1998)*, 1998. Not included in the official proceedings.

[Bow04] Jonathan P. Bowen. Virtual Library - Formal Methods. `http://vl.fmnet.info`, 2004.

[Cha98] Pierre Chapront. Ada+B: The Formula for Safety Critical Software Development. In Lars Asplund, editor, *Proc. Reliable Software Technologies: Ada Europe (RST'1998)*, pages 13–18. Springer, Berlin, LNCS 1411, 1998.

[CW96] Edmund M. Clarke and Jeanette M. Wing. Formal Methods: State of the Art and Future Directions. *ACM Computing Surveys*, 28(4):626–643, 1996.

[Gau01] Marie-Claude Gaudel. Testing from Formal Specifications, a Generic Approach. In Dirk Craeynest and Alfred Strohmeier, editors, *Proc. Reliable Software Technologies: Ada Europe (RST'2001)*, pages 35–48. Springer, Berlin, LNCS 2043, 2001.

[Gla96] Robert L. Glass. Formal Methods are a Surrogate for a More Serious Software Concern. *IEEE Computer*, pages 19–19, April 1996.

[GR01] Martin Gogolla and Mark Richters. Expressing UML Class Diagrams Properties with OCL. In Tony Clark and Jos Warmer, editors, *Advances in Object Modelling with the OCL*, pages 86–115. Springer, Berlin, LNCS 2263, 2001.

[Hal90] J. Anthony Hall. Seven Myths of Formal Methods. *IEEE Software*, 7(5):11–19, September 1990.

[HB96] C. Michael Holloway and Ricky W. Butler. Impediments to Industrial Use of Formal Methods. *IEEE Computer*, pages 25–26, April 1996.

[KDGN97] John C. Knight, Colleen L. DeJong, Mathew S. Gibble, and Luís G. Nakano. Why Are Formal Methods Not Used More Widely? In C. Michael Holloway and Kelly J. Hayhurst, editors, *4th NASA Langley Formal Methods Workshop*, pages 1–12, 1997. NASA Langley Research Center, Report CP 3356.

[OMG03] OMG, editor. *OMG Unified Modeling Language Specification, Version 1.5.* OMG, March 2003. OMG Document `formal/03-03-01`, `www.omg.org`.

[RG00] Mark Richters and Martin Gogolla. Validating UML Models and OCL Constraints. In Andy Evans and Stuart Kent, editors, *Proc. 3rd Int. Conf. Unified Modeling Language (UML'2000)*, pages 265–277. Springer, Berlin, LNCS 1939, 2000.

[RG01] Mark Richters and Martin Gogolla. OCL - Syntax, Semantics and Tools. In Tony Clark and Jos Warmer, editors, *Advances in Object Modelling with the OCL*, pages 43–69. Springer, Berlin, LNCS 2263, 2001.

[Sch00] Wolfram Schulte. Why Doesn't Anyone Use Formal Methods. In W. Grieskamp, T. Santen, and B. Stoddart, editors, *Proc. Int. Conf. Integrated Formal Methods (IFM'2000)*, pages 297–298. Spinger, Berlin, LNCS 1945, 2000.

[TRF03] A. Toval, V. Requena, and J.L. Fernandez. Emerging OCL Tools. *Software and Systems Modeling*, 2(4):248–261, 2003.

[vL01] Axel van Lamsweerde. Building Formal Requirements Models for Reliable Software. In Dirk Craeynest and Alfred Strohmeier, editors, *Proc. Reliable Software Technologies: Ada Europe (RST'2001)*, pages 1–20. Springer, Berlin, LNCS 2043, 2001.

[WK03] Jos Warmer and Anneke Kleppe. *The Object Constraint Language: Precise Modeling with UML*. Addison-Wesley, 2003. 2nd Edition.