Formal Techniques for OO Software Development (PANEL)

Dennis de Champeaux, *HP-Lab Palo Alto*, (moderator)
Pierre America, *Philips Research Laboratories*
Derek Coleman, *HP-Lab Bristol*
Roger Duke, *University of Queensland*
Doug Lea, *Syracuse University & SUNY-Oswego*
Gary Leavens, *Iowa State University*

Co-organizer: **Fiona Hayes**, *HP-Lab Bristol*

Background

In this panel, we discuss the relevance of formal techniques for applying object-orientation.

The object-oriented paradigm is currently broadened from the programming realm to cover the full development life cycle, including (domain) analysis and design.

These extensions are driven by the demands of large system development. Delivering huge OO software systems routinely and cost effectively is a significant challenge. To quote Ed Yourdon: "A system composed of 100,000 lines of C++ is not to be sneezed at, but we don't have that much trouble developing 100,000 lines of COBOL today. The real test of OOP will come when systems of 1 to 10 million lines of code are developed."

Scaling up seems to require increasing the precision of the semantics of the languages/ notations used by a team.

Large, mission critical projects may have to demonstrate that a target system satisfies its specifications provably. This suggests having, at least, formal semantics for the specification language.

When a target system contains an abundance of parallelism, we face the problem of validation. Errors can be nearly impossible to duplicate in a concurrent setting. Transformations that preserve meaning can assist the validation. They require precise semantics in their domain of operation.

Relational databases are well understood. They can be accessed out of "any" programming language. OO database vendors wants to offer the same service. Relational database theory is soundly grounded in mathematical theories like the (tuple) relational calculus. Similar formal foundations may be required for objects.

Can formal techniques play a role to solve the above mentioned issues? The panel will contrast short term feasibility and relevance against what to expect in the medium and longer term.

To help focus the discussion, we look at the following specific topics:

- Will using OO throughout the life cycle increase the level of formality in comparison to what has been achieved in the structured paradigm?

- Will formal techniques in OO make the testing/ validation/ verification of a target system against its requirements easier by orders of magnitude in comparison with current practice?

- Will formal techniques facilitate the development of sizable libraries (at the conceptual, design and code level)? If so, can we expect massive reuse, and thus a speed up of the development process by orders of magnitude?

Pierre America

Object-oriented programming has originally developed without a strong basis in mathematical formalism. For many people it even seemed to offer a solution to the software crisis without the need for all that complicated math. However, recently there is a rapidly increasing interest in applying formal techniques to object-oriented programming.

There are several ways in which formal techniques could help in object-oriented software development:

- Describing the basic concepts of our languages and systems in a formal, mathematical way can dramatically enhance our understanding of them and in such a way lead to better languages and systems. This applies particularly to databases, parallelism, typing, and support for the early phases of development (e.g., requirements specification). Here it is not necessary that the individual programmer has detailed knowledge of all applicable formal techniques, but that the designers of the language or model take them into account, thus coming up with a better environment for the programmer to work in.

- Formal techniques can support the development of methods for software development, which can then be taught to the programmers in a less formal, but nevertheless rigorous way. This is what happened to high school mathematics, but also to well-known techniques such as pre- and postconditions and invariants.

- In order to be able to reuse software components, it is necessary to describe what exactly they do, without referring to the code (which is often intentionally unavailable). Formal specifications have definite advantage over natural language here because they are unambiguous (I know about 50 different things that could each be called a 'stack') and because they can be processed by a machine (which opens perspectives for automatic search in component libraries, for example). Even if the actual specifications used are not (completely) formal, formal techniques could help us to develop a framework for specifying components independently of the code that implements them.

- Finally, there is of course the possibility of full formal verification of critical software. It will certainly still take a long time before this is a routine matter, but to a limited extent these techniques can already be used now. However, I think that this aspect has been overemphasized in research on formal techniques.

While the potential is clearly present, a lot of problems still have to be solved before the above possibilities can be exploited fully. Some of these problems are specific for object-oriented programming (for example, the semantic treatment of inheritance), but others have been around for a long time and either have been shied away from (e.g., formal description of dynamically changing pointer structures) or have proved to be so difficult that even after decades of active research there is no final solution yet (e.g., specification and verification of concurrent systems). Therefore there is no hope that formal techniques will shortly lead to a revolution in the software development process, but I am sure that they will make all the difference in the long run. Nevertheless, experience in my company shows that even on a short term, the right use of formal techniques supported by the right tools can make a substantial contribution to the speed and accuracy of the development process. And that is

what it is all about: formality is not a goal in itself, but it is only useful as a means towards more efficient and more reliable software development.

Derek Coleman

Motto: What's Formal Methods got to do with Object-oriented Development?

The success of the object-oriented approach has gained much attention during the last decade. However the success is at the level of small team developments. Industry has learned the hard way that large scale efforts are not straightforward. We are beginning to remember what we used to know! The laws of software engineering still hold - software development must be a systematic and managed process. **Objects are not a panacea.**

Yourdon's test[1] is a measure of the problem that faces the new paradigm. One of the responses is the upsurge in interest in object-oriented analysis and design and CASE tools. Currently there is a deluge of object-oriented analysis and design methods and CASE tools.

A method is essentially a set of notations together with a strategy, and heuristics, for deploying them. The best of the new methods have effective strategies and contain useful heuristics, but they are characterized by too much emphasis on naturalness of expression and intuition. When it comes to notations *ad-hoc*-ery is the order of the day. Everybody's powerful feature is included and every difficult issue is ignored. Whichever method you choose, you can be sure that the models that you develop are built on sand and hence any supporting tool can be little more than a diagram editor.

In order to measure up to Yourdon's test, develop safety-critical software, or write large concurrent software, the methods must use *notations that have a semantics.* This is where **formal methods are relevant.** Despite the name, formal methods has relatively little to say about methods - it is all about formal notations. The notations are wide and varied. Specification languages like Z, VDM and HP-SL provide precise and abstract descriptions of software. Algebraic languages like OBJ and Axis show how modularity and executability can be combined to provide design time prototyping and testing. Higher order logic specification languages, like HOL, have proved effective in verifying designs.

I believe that if formal notations were incorporated into object-oriented methods then we could expect:

- Improved analysis and design methods that produce coherent models capable of being validated against requirements.
- Rigorous object-oriented methods of development for use on safety-critical applications.
- Object-oriented CASE tools which can check the semantics of models.

The panel statement asks whether using the object-oriented approach throughout the life cyle will increase the level of formality? No, it will not. Structured methods have not led to the introduction of formal methods. It will require a conscious effort to combine the naturalness of objects with the precision of formality in order to improve the quality of object-oriented techniques. Until such a fusion takes place I am pessimistic about the prospects for large-scale object-oriented software development.

Roger Duke

Object orientation can be viewed as a natural progression in the trend towards increased formality in software design. Issues like reusability, inheritance and subtyping that are integral to object orientation encourage the sound formal design of systems, and it can be argued that it is precisely because of this that object orientation is becoming so important.

But are we now in a position where the development, production and maintenance of software systems can be considered an engineering discipline?

[1] Can we build object-oriented systems that are composed of millions of lines of code?

Is the term *Software Engineering* an accurate reflection of our discipline, or merely some fraudulent misnomer?

Although system design is central in any formal development, it is only part of the story: methods for formal specification and verification need to be combined with object-oriented design, and the whole structure integrated within a sound refinement theory. At present we have a collection of somewhat disjoint theoretical results. A major problem that remains is the weaving of these results into a unified, cohesive and *practical* engineering discipline that is formally based.

There is good evidence that the object-oriented paradigm can suggest to us how to proceed.

- Object-oriented specification techniques are being developed. Central to this development is the realization that specification can be strengthened by incorporating aspects of object-oriented design, quite contrary to the conventional wisdom that separation of concerns demands that the issues be divorced.
- Refinement within object-oriented systems can be related to subtyping and realized by (restricted) inheritance, i.e. inheritance itself suggests a possible approach to refinement.
- As object-oriented systems are constructed by composing underlying objects, this compositional structure strongly suggests how verification proofs could also be structured.

Hence the prognosis is good. For those willing to dream of a future Utopia there is every reason to believe that object orientation can be combined with other formal methods to produce techniques that will dramatically assist the software engineer throughout the software-cycle.

For those more concerned with taking stock of where we are at the present time, we already have a collection of (somewhat incomplete) formal object-oriented techniques that, even although far from perfect, can nevertheless help us to construct more reliable software systems whose behavior can be predicated and guaranteed with some certainty. Should we be using these formal techniques? Given the increasing importance of software correctness and reliability we're damn fools if we don't.

Doug Lea

People have grounds to be naively optimistic about the prospects for coupling formal methods with OO design and programming.

Many everyday aspects of OOD/OOP represent "informal formal methods", that are analogous to those found in various formal specification systems. These include the use of abstract classes to declare behavior in an implementation-independent fashion, the use of inheritance to indicate subtype relations, and reliance on well-understood abstractions like `Sets`, `Sequences`, and `Maps`.

The effects of such practices may be enhanced by the creation of formal methods that extend the semantic power of these aspects of OOD/OOP, by supporting the expression of predicate-based constraints, axioms, and invariants that allow fuller specification of behavior than is now possible in most common OO languages. Languages and support tools need to evolve to accommodate behavioral specification constructs that are, ideally, as natural and successful as the OO constructs and practices they extend and modify.

Use of formal methods need not be separated from ordinary successful OO software development activities. Efforts to integrate stronger specification methods into OO languages themselves, even when this results in loss of expressiveness and/or verifiability are more likely to be *used* in everyday development than are isolated formalisms.

While verification is an important aspect of formal approaches to software development, automated verification of OO software consisting of the kinds of mutable, aliased, and/or concurrent objects commonly found in OO programs remains a distant goal. However, researchers and practitioners in non-OO contexts have found that the use of formal methods by designers and programmers improves reliability, testability, and productivity, because of the precision and completeness of re-

quirement and design specifications formal methods encourage. It does not seem farfetched to believe that the same will hold true for OO formal methods, which already tend to outperform structured methods on these qualities.

While formal methods themselves will not automatically cause greater reuse, their pragmatic role in increasing "consumer confidence" is an important ingredient in making good on the promise of OO methods to result in massive reuse and productivity improvements. In order to use off-the-shelf components with confidence, clients require detailed semantic specifications of the components, optimally along with evidence that implementations meet those specifications. This, in turn has the positive effect of enhancing the well-specifiedness of large systems that use many such components.

Gary Leavens

Formal techniques can aid in the development of high quality application frameworks and libraries of reusable modules (such as classes). It is a mistake to concentrate on getting more code faster, because poorly designed, bug-ridden code will not be reused. If you accept the premise that quality is essential for reuse, then you are led inexorably to formal techniques. Quality software is well documented, easy to understand, etc. The more careful you try to be about such aspects of software, the more you will use formal techniques.

For example, to develop a reusable application framework, you should reuse it yourself. Hence you are constantly playing both developer and client. Specifications that say what properties of software a client can rely on are crucial for keeping these roles straight. You must be sure to program client code from these specifications, so that you do not use your developer's knowledge as a client. A formal verification can guarantee, for example, that client code only relies on a method's specification, not its implementation details. Informal techniques are too easily supplemented by your intuition as a developer.

Object-oriented design makes consistent use of abstraction. Data abstraction helps you separate implementation from observable behavior. Supertype abstraction, letting supertypes stand for their subtypes, helps you separate interesting behavior from non-interesting behavior. These kinds of abstraction by specification are useful not only in programs but also in formal specification, verification, design, and testing. For example, you can verify the effect of a message send, based on the specification of the receiving expression's static type, without knowing the receiving object's dynamic type or what class implements it. That is, you use data abstraction to ignore implementation detail, and supertype abstraction to ignore some of the observable behavior of subtypes.

In design, you can use subtyping as a more general version of data type refinement. You can create a subtype to embody a design decision, and to limit the parts of a design that depend on that decision. You can also use subtype relationships to organize the types in a framework or library in a way that is suited to verifiers and clients, since subtype relationships are based on observable behavior. Tests can also be organized in layers based on subtype relationships, since tests for a subtype can inherit tests from a supertype's test suite.

The real problem in validating software against requirements is finding the "right" requirements, since one cannot formally validate software against informal requirements. Exploratory programming has been useful when requirements are fuzzy, but it can be expensive to construct a prototype and throw it away. A better way to find the right requirements would be exploratory specification. In exploratory specification, you would formally specify fuzzy aspects of the system as many times as needed to firm up the requirements. Then you could prototype just those aspects of the system.