Spying on Components: A Runtime Verification Technique

Mike Barnett and Wolfram Schulte
Microsoft Research
One Microsoft Way
Redmond WA, 98052-6399, USA
{mbarnett,schulte}@microsoft.com

ABSTRACT

A natural way to specify component-based systems is by an *interface specification*. Such a specification allows clients of a component to know not only its syntactic properties, as is current practice, but also its semantic properties. Any component implementation must be a behavioral refinement of its interface specification. We propose the use of executable specifications and a runtime monitor to check for behavioral equivalence between a component and its specification. Furthermore, we take advantage of the COM infrastructure to perform this kind of *runtime verification* without any instrumentation of the implementation, i.e., without any recompilation or re-linking.

1. INTRODUCTION

We believe that component-based programming needs formal specifications at the interface level. Currently there are standardized ways to formally specify the syntactic properties of a component, for example, by type libraries or IDL files for COM components [7]. However, the proper mechanism for specifying semantic properties is still an open research topic. Clearly, clients of a component, whether they are human or other software components, require some way of understanding the behavior of a component. Natural language descriptions, while valuable, are often incomplete or ambiguous and are in any case limited to human consumption.

Even if there was agreement on a particular specification technique, there is still the problem of ensuring that a particular component does indeed implement its specification. We propose an answer to the first problem and a technique that partially addresses the second problem.

Our approach for specifying components is to use AsmL to write an executable specification at the highest level of abstraction that defines the behavior of a component as seen through its interface by a client. AsmL is an industrial-strength specification language we have developed at Microsoft Research. Based on the theory of Abstract State Machines (ASMs) [16], it allows the writing of operational specifications at any given level of abstraction. Using it, we have built models of real-world components, like intelligent devices, internet protocols, debuggers and network components [2, 14]. Because ASMs have a formal semantics, an AsmL specification is itself a formal specification.

For the second problem, we use AsmL's native COM connectivity and the COM infrastructure to dynamically monitor the execution of a component. By checking for behavioral equivalence between the component and its concurrently executing specification we ensure that, during a particular run, the component is a behavioral refinement of its specification, i.e., a behavioral subtype [22].

There are two major issues that we do *not* address in this paper: non-deterministic specifications and callbacks. Both are crucial elements of component-based specification and we have developed solutions for both of them, but they are beyond the scope of this paper. Although our system is implemented for COM components, it applies to any component technology that uses dynamic linking.

The paper is organized as follows. Section 2 gives an overview of AsmL. Section 3 explains how to use AsmL to write an interface specification. Then in Section 4 we explain our technique for runtime verification. In Section 5 we describe some initial experiments we have conducted within Microsoft. An overview of similar approaches is discussed in Section 6; Section 7 summarizes, presents limitations, and describes future work.

2. AsmL

We write executable specifications of components in the Abstract State Machine Language (AsmL). The language is based on the theory of Abstract State Machines [16]. It is currently used within Microsoft for modeling, rapid prototyping, analyzing and checking of APIs, devices, and protocols.

The key aspects which distinguish AsmL from other related specification languages are:

- it is executable,
- it uses the ASM approach for dealing with state,
- it has a full-fledged object and component system,
- it supports writing non-deterministic specifications.

Our web site [13] contains a complete description as well as an implementation that is freely available for non-commercial purposes.

OOPSLA 2001 Workshop on Specification and Verification of Component-Based Systems Oct. 2001 Tampa, FL, USA

Because AsmL has native COM connectivity (the next release will also be integrated into the .NET framework), one can not only specify components in AsmL and simulate them but also substitute low-level implementations by high-level specifications. This substition allows heterogeneous systems to be built, partly developed using standard programming languages and partly using executable specifications. It is also crucial for implementing runtime verification without the need for instrumenting the implementation.

Although not shown in this paper, non-determinism is one of the key features of AsmL. It allows designers to clearly mark those areas where an implementation must make a decision. In AsmL, non-determinsm is restricted; you can choose or quantify only over bounded sets [5].

AsmL specifications are *model programs*: they are operational specifications of the behavior expected of any implementation. Thus, they provide a *minimal model* by constraining implementations as little as possible. There are three main properties of AsmL that support this.

1. The ASM notion of *step* allows the specifier to choose an arbitrary granularity of sequentiality. Within a single step, all updates (assignment statements) are evaluated in parallel; locations (variables) are written in one atomic transaction at the end of the step. Using the maximal step size means that no unnecessary sequencing is forced on the implementer. An implementation is free to choose an evaluation order consistent with efficiency considerations.

2. Non-deterministic (bounded) choice is a basic construct in the language. Although non-determinism is undesirable in an implementation, non-deterministic specifications allow implementations to make the correct engineering decisions. For instance, a specification might say that any element satisfying certain conditions can be returned from a collection where the implementation might make a particular choice based on the efficiency of searching the data structures that are employed.

3. High-level data structures and programming constructs allow a specification to be expressed in ways that might not be acceptable when efficiency is the primary concern. For instance, a specification might not bother to normalize a data structure, but instead re-organize and manipulate it each time it is accessed.

In general, the primary goal of a specification is to be as clear and understandable as possible; the goal of an implementation is to meet engineering considerations such as execution time, storage efficiency, etc.

Compared to an implementation language such as C++, we have found AsmL specifications to be an order of magnitude more compact. While part of this is due to the advantages offered by any higher-level notation, some part is caused by the specific features of AsmL enumerated above.

3. INTERFACE SPECIFICATIONS

Figure 1 presents a small example that we use throughout. It is not COM-specific. Although written in AsmL, it corresponds exactly to an interface expressed in IDL. AsmL makes implicit the fact that COM methods also return a status value in addition to whatever other values they return; compiler-generated code handles that automatically.

```
interface ICanvas
  createFigure(...) as IFigure

interface IFigure
  getColor() as Color
  setColor(c as Color)
  getBorder() as IBorder

interface IBorder
  getWidth() as Integer
  setWidth(i as Integer)
```

Figure 1: Example Interfaces: Syntax Only

```
interface ICanvas
  createFigure(c as Color, ...) as IFigure =
    new IFigure(c, ...)

interface IFigure
  var color as Color
  border as IBorder = new IBorder(3)
  getColor() as Color = color
  setColor(c as Color) = color := c
  getBorder() as IBorder = border

interface IBorder
  var w as Integer
  getWidth() as Integer = w
  setWidth(i as Integer) =
    if i < 0 then throw Exception(...) else w := i
```

Figure 2: Example Interfaces: Semantics

The example provides interfaces for a component-oriented drawing program: a client interacts with a root interface, *ICanvas*, to create and manipulate geometric figures, which support the interface *IFigure*. Each figure has a nested object, a border, which supports the interface *IBorder*. That is, a component supporting the *IFigure* interface also must be able to provide a reference to an *IBorder* interface. Whether this reference is actually to a separate component, or just a different interface on the same component is exactly the kind of underspecification that component-based programming encourages.

The method *createFigure* returns a reference to the *IFigure* interface on the figure that is created. A figure's border is created with some default attributes; the attributes can be changed later through calls to methods such as *setWidth*. Note the syntax of the interface definitions alone allows data values and interface references to be distinguished.

An example AsmL specification for this interface is shown in Figure 2. It is written as a model program, as opposed to a set of pre- and post-conditions (although AsmL does provide also that style of specification). It is a particularly trivial model; this is good — such a trivial component should not have a complicated specification.

Note that the method *setWidth* throws an exception if the argument is less than zero. The exception must belong to

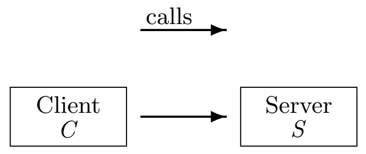

Figure 3: A client-server architecture

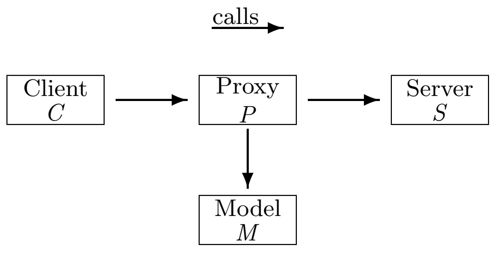

Figure 4: Proxy Architecture

some interface, but we do not show it.

4. RUNTIME VERIFICATION

A component that interacts with an implementation of *ICanvas* is a *client*, while the implementation is a *server*; together their architecture is shown in Figure 3. We refer to the client program as C and the implementation of the server as S. The important feature implied by such an architecture is that the client is completely unaware of the identity of the server component. C is aware solely of the functionality provided through whatever interfaces are supported by S. This is the crucial feature we rely on for implementing our runtime verification.

To enable the AsmL specification to spy on the interactions between C and S, we insert a component, P, which operates as a proxy, as shown in Figure 4. Using a proxy allows the interaction of the client C and the server S to be observed without having to instrument (i.e., modify) either component. The proxy forks all of the calls made from C to S so that they are delivered to the AsmL specification or model, M. From now on, we use the letters C, S, M, and P to refer to the client, server, model, and proxy, respectively.

Inserting a proxy is easily accomplished for COM components [7]. Clients can initiate access to a COM component only by making a request to the operating system. That request can be intercepted either with or without the client's cooperation. As long as the value returned to the client is a valid interface reference, the client is unable to distinguish whether the reference is to the actual implementation, S, or to our proxy, P. In fact, it is this property of COM that allows transparent access to COM components that are not local to the machine on which the client is executing.

Runtime verification means that from the client's point of view, the observed behavior of the model is indistinguishable from that of the server, i.e., they are *behaviorally equivalent*. Because this is a dynamic check, it means they are equivalent only on the observed behavior; ideally the specification allows more behaviors. An implementation restricts its behavior, usually for reasons of efficiency.

When runtime verification uncovers a difference in behavior between the specification and the implementation, there is no *a priori* way to know which is "wrong" (unless one assumes that the specification is always correct...). We are unaware of any method for creating perfect specifications; one can only hope to use a specification language that supports a layered approach. Engineering practice has shown the importance of separating unrelated concerns in order to focus on the proper details at issue. We have tried to ensure AsmL is such a language.

4.1 The Proxy P

All method calls between C and S are intercepted by P. As far as C is concerned, it is accessing the functionality provided by S and is unaware of either P or M. P manages the concurrent execution of M and S; it forks every call so that they are delivered to M as well as S. P compares the results from both components, checking at each interface call that they agree in terms of their success/failure codes as well as any return values. (In our examples, we do not explicitly show the checks for the success or failure of the methods.) As long as they are the same, the results are delivered to C. Otherwise S and M are not behaviorally equivalent; the discrepancy is made evident to an observer of the system.

We create P automatically from the definition of the interfaces that are used between C and S. The correct operation of P relies on two properties of object references in S that allow them to be used as identifiers.

1. They must be *stable*: an object reference returned to C maintains its identity in S. The client C can always use that reference to refer to the same object.

2. They can be tested for equality: a reflexive, symmetric operation allows P to distinguish different objects.

We believe both of these properties to be reasonable and easily met; we mention them only to be explicit about our dependencies.

4.2 Verifying Data

For methods that return atomic data values, runtime verification is comparatively simple. P maintains a global table *map*, which stores object references created in P to pairs of corresponding model and server object references:

```
map as Map of Object to (Object * Object)
```

The datatype *Map* in AsmL is an associative array, i.e., an array whose indices do not have to be integers. Initially this table contains just one entry: the reference of the root object of P along with the tuple containing the references to the roots of M and S. This entry is created when C first connects to P. As object references are returned to the client, the map is kept current, as explained in Section 4.3.

Thus when the client uses an interface reference of type *IFigure* to call *getColor*, it is really calling the method on an instance of a class, *PFigure*, defined in P. The behavior of *PFigure.getColor* is shown in Figure 5. The table *map* is consulted to retrieve the interface references to M and S. Each is an interface reference to an object implementing the interface *IFigure* in M and S, respectively. The method *getColor* is called in each of the components and their return

```
class $PFigure$ implements $IFigure$
  $getColor()$ as $Color$ $=$
    let $(M, S) = map(me)$
    let $m = M.getColor()$
    let $s = S.getColor()$
    if $s \neq m$ then
      throw $Exception(\ldots)$
    else
      return $s$
```

Figure 5: *PFigure.getColor*

values are then compared to guarantee that M and S remain equivalent, from the perspective of the client.

Consider the similar method *setWidth* that would be defined on an instance of a class *PBorder*. If the client called it with a negative argument, then M would throw an exception. In such a case, $S.setWidth$ should also throw a subtype of the same exception type.

4.3 Verifying Objects

The simple scheme outlined in Section 4.2 breaks down when a method returns an interface reference. For instance, in our example, the methods *createFigure* and *getBorder* both return interface references. Consider the situation when a client calls *createFigure* (which is probably the first method the client will call). Our proxy, P, calls the method on both the implementation and the model. Both M and S will return to P a created object internal to the respective components; the objects must support the *IFigure* interface.

One problem is that there is no way for P to make an equality test between the references returned from M and S. That is, it cannot decide *at this time* whether or not the two figures are the same. That can be decided only as operations returning simple data (such as *getColor*) on those figures are invoked.

Another problem is that P needs to return a reference (to an object supporting the *IFigure* interface) to the client. If the interface reference from S is returned directly to C, then P will no longer be able to monitor the communication between C and S. C may use that interface reference to make further method calls and those calls would go directly to S.

To solve both problems, P creates a a new local object, p, from the class *PFigure*. P installs the pair of objects returned from M and S in the global table *map*, indexed by p. Instead of returning either the reference from M or S, it returns a reference to the local object p. Then, when the client calls *getColor* on p, it is executing the method shown in Figure 5.

In this way, all interface references from S are *spoofed*. (This is the standard way marshalling proxies are created for remote interfaces in COM [7].) Returning the interface reference to the local object means that all future calls can be monitored.

Now, consider the case when an interface reference is *not* new; say it is a reference that has been returned from S in some previous call. For instance, if *getBorder* is called more than once on the same figure, the same reference will be returned. So if S returns an interface reference, s_1, then M

```
class $PCanvas$ implements $ICanvas$
  $createFigure(\ldots)$ as $IFigure$ $=$
    let $(M,\ S) = map($me$)$
    let $m = M.createFigure(\ldots)$
    let $s = S.createFigure(\ldots)$
    if $(m =$ nothing$)$ and $(s =$ nothing$)$ then
      return nothing
    else
      let $p = checkObjects(m, s)$
      if $p =$ nothing then
        let $p' =$ new $PFigure()$
        $map(p') := (m, s)$
        return $p'$
      else
        return $p$
```

Figure 6: *PCanvas.createFigure*

```
$checkObjects(m$ as $Object,\ s$ as $Object)$ as $Object$ $=$
  if $(m =$ nothing$)$ or $(s =$ nothing$)$ then
    throw $Exception(\ldots)$
  elseif $\exists\ p \in domain(map)$
            where $map(p) = (m, s)$ then
    return $p$
  elseif $\exists\ p \in domain(map)$
           where $first(map(p)) = m$
                  or $second(map(p)) = s$ then
    throw $Exception(\ldots)$
  else
    return nothing
```

Figure 7: *checkObjects*

must also have returned an interface reference, m_1. Again, there is no way to know if the two interface references refer to two "equal" components: equality cannot be decided between them.

However, the references must have been seen together as a pair the previous time; this is where we assume the stability of the interface references. If the two returned references form such a pair, then there is a local object, p, such that $map(p)$ is the pair (m_1, s_1). Then p is the spoof for the pair and should be returned to C. Otherwise, there is some other pair in the map (m_2, s_1), indexed by another local object p'. (Remember the assumption is that s_1 has been seen before, i.e., returned from S at some earlier method invocation.) This is enough evidence to know that M and S are not behaviorally equivalent because they are not responding in the same way to the same method call. The symmetric argument handles the case when m_1 has been seen by P before.

All of these possibilities are illustrated in P's method for *createFigure* as shown in Figure 6. The logic that decides the correspondence (or lack thereof) between the returned interface references is enapsulated in the method *checkObjects*, which is defined in Figure 7. This explains how the entry in the table retrieved in Figure 5 was initially created.

5. EXPERIENCES

Within Microsoft, we have used AsmL for runtime verification in two case studies on existing product components. Since they already existed, we reverse-engineered an AsmL model from the available documentation, discussions with the responsible product group, and (self-imposed) limited access to the source code. We did not want to re-implement the current components, but wanted to have a true n-version system. Both components are of medium-size: between 50 and 100 thousand lines of code (LOC).

The first case study was partially described in [2]. We created a model of the Debug Services component for the .NET Runtime. The Debug Services control the execution of a .NET component in the runtime; a debugger is a client that requests the installation and removal of breakpoints, etc. (In turn, a person executing a debugger is a client of the debugger.) Our model was less than 4K LOC. The published case study is more concerned with describing the methodology for creating the specification. In the course of performing runtime verification, we encountered a violation of the Debug Services protocol. In conversations with the product team, it turned out that there was an unresolved ambiguity in the meaning of one method when used to respond to a callback. While it could not be considered a major bug in any sense, it did make them realize that they had never decided how to resolve the ambiguity even though they had held meetings about it. Had they been using runtime verification, the problem would not have been able to lie hidden for so long.

For our second case study we modeled the Network Configuration Engine that is part of the Windows operating system. The engine is responsible for maintaining a database of installed network drivers and the network paths that exist between them. We wrote the specification only from the documentation; it ended up being about 2K LOC. We performed runtime verification using an automated test suite provided by the product group and again found a discrepency between the model and the implementation. For one particular method, a flag is used to choose between two different behaviors. However in the real implementation there had originally been three different behaviors and the one that was removed was different from the one that was removed from the documentation. This demonstrated the usefulness of having a specification as documentation: had it been used during the development process, the documentation would have been guaranteed to be consistent with the implementation.

6. RELATED WORK

The need to specify and check components is widely recognized (cf. [26]). However there is neither a standard way to specify components nor any standard for checking an implementation's conformance with its specification.

In a recent book, Leavens and Sitaraman [19] summarize the current approaches for specifying components formally. In that book, Leavens and Dhara [20] use the specification language JML to specify Java components. As we do, JML uses model programs in addition to pre- and post-conditions. Our approaches are very similar, but JML is restricted to specifying Java, while AsmL can be used with any programming language. Müller and Poetzsch-Heffter's [25] article in the same volume also concerns the specification of interfaces, but with pre- and post-conditions. Their main concern is the verification of frame properties, i.e., controlling the modifications a method can make.

In Edwards et al. [9], an architecture is proposed for deriving wrappers for any class implementing an interface that is enriched with pre- and post-conditions. Human intervention is required to map the concrete state of the class to the abstract state used in the interface specification. The advantage of our approach is that the operations on the abstract state are independent of the concrete state, so an AsmL specification can check any implementation. However, the use of an abstraction function means that discrepencies can potentially be discovered earlier than by checking behavioral equivalence as we do.

Jonkers, working at Phillips, is also working on interface specifications [18]. In their work on Ispec, they use transition systems to provide the semantics for interface specifications. However they don't try to execute the model in isolation or run it in parallel with the implementation. Instead they want to generate black-box tests.

Besides JML, there has been a lot of work on using assertions to specify Java interfaces, e.g., Contract Java [11, 12], iContract [8], and Jass [4]. And of course, Eiffel [23, 24], uses pre- and post-conditions to specify components. However, these do not introduce model programs as we do.

Closer to our work on runtime verification is the work on program checking as proposed by Blum and Wasserman [6]. They argue that it is often much easier to write a program that checks whether a result is correct, than to prove the algorithm correct that produces the result. For example, it is difficult to factor an integer, but, given x and y, it is trivial to determine whether or not y is a factor of x. In our case the checker is the specification.

Using this idea, Antoy and Hamlet [1] propose the use of algebraic specifications to specify software. Algebraic specifications use high level data structures, thus solving one of the aforementioned problems of pre-/post-conditions. The price is that when checking the implementation against the specification one needs abstraction. Their system is able to run the executable specification (in fact it is a rewrite system) in parallel with the implementation in C; similar to our framework, they check the results on the method boundaries. They include a comprehensive review of similar work; we do not repeat it here. But due to the restricted nature of algebraic specifications, they cannot deal with state or with object identities (without a lot of coding).

Another similar project is the SLAM project by Herranz-Nieva and Moreno-Navarro [17]. They developed a new specification language and define class operations with pre-/post-conditions. The resulting specifications are translated to C++; part of the pre-/post-conditions are compiled to Prolog. Using a bridge between C++ and Prolog, the Prolog clauses are used as assertions during runtime. Results are speculative, since the project is in the early stages of development.

While not specifically relating to interface specification, Erlingsson and Schneider [10] have also developed a method for injecting a runtime monitor into programs to enforce security properties. In their examples, the monitors are derived from finite automata and so are consequently limited. The transitions of the automata must be triggered by events that are observable at the level of machine code. This is appropriate for the security properties they check, but are not suited for checking interface properties.

Instead of performing checks at runtime, there has been much work using *static analysis* to prove general properties about a program. While it provides a more general result that is true of any execution of the program, the limitations of program analysis enforce a consequent weakening of the set of properties that can be checked. Perhaps the most well known static program checker is ESC/Java [21].

7. CONCLUSIONS

We have presented a specification method for interfaces that allows a component implementing the interface to be run concurrently with its specification with no need for re-compiling, re-linking, or any sort of invasive instrumentation at all. While runtime verification does not prove that the component is correct (with respect to its specification), it does guarantee that, for that particular trace, the component is a behavioral subtype of its specification. For systems that are not amenable to current formal verification technology, this may be the highest degree of formal proof possible. To be useful in real-world applications, formal specifications must provide benefit within the existing development processes. Runtime verification can be used as part of current testing techniques, whether directed or ad-hoc.

We have used our methods to model two medium-sized components within Microsoft and performed runtime verification during user scenarios as well as in the context of testing using an automated test suite. Both times we have been able to find discrepencies between the actual component and its specification.

While this presentation has been restricted to deterministic specifications and systems that do not make callbacks, these burdensome qualifications are addressed in a more complicated scheme [3]. Unfortunately, this scheme is subject to exponential worst-case behavior. We are developing a new system that will be integrated into the .NET runtime, which does not suffer from this drawback.

Our specification language, AsmL, allows other opportunities which are beyond the scope of this paper. For instance, we have used it for early prototyping and test-case generation [15].

We believe that runtime verification shows promise in providing automated support for keeping a specification alive and for ensuring that an implementation correctly implements its specification.

8. REFERENCES

[1] Sergio Antoy and Richard G. Hamlet. Automatically checking an implementation against its formal specification. *Software Engineering*, 26(1):55–69, 2000.

[2] Mike Barnett, Egon Börger, Yuri Gurevich, Wolfram Schulte, and Margus Veanes. Using Abstract State Machines at Microsoft: A case study. In *Abstract State Machines: Theory and Applications*, volume 1912 of *LNCS*, pages 367–379, Berlin, Germany, March 2000. Springer-Verlag.

[3] Mike Barnett, Lev Nachmanson, and Wolfram Schulte. Conformance checking of components against their non-deterministic specifications. Technical Report MSR-TR-2001-56, Microsoft Research, June 2001. Available from `http://research.microsoft.com/pubs`.

[4] Detlef Bartetzko, Clemens Fischer, Michael Möller, and Heike Wehrheim. Jass — Java with Assertions. http://semantik.informatik.uni-oldenburg.de/~jass/doc/index.html.

[5] A. Blass, Y. Gurevich, and S. Shelah. Choiceless Polynomial Time. *Annals of Pure and Applied Logic*, 100:141–187, 1999.

[6] Manuel Blum and Hal Wasserman. Software reliability via run-time result-checking. *Journal of the ACM*, 44(6):826–849, November 1997.

[7] Don Box. *Essential COM*. Addison-Wesley Publishing Company, Reading, Massachusetts, 1998.

[8] A. Duncan and U. Hölze. Adding contracts to Java with handshake. Technical Report TRCS98-32, University of California at Santa Barbara, December 1998.

[9] Stephen H. Edwards, Gulam Shakir, Murali Sitaraman, Bruce W. Weide, and Joseph Hollingsworth. A framework for detecting interface violations in component-based software. In P. Devanbu and J. Poulin, editors, *Proceedings: Fifth International Conference on Software Reuse*, pages 46–55. IEEE Computer Society Press, 1998.

[10] Ulfar Erlingsson and Fred B. Schneider. SASI enforcement of security policies: A retrospective. Technical Report TR99-1758, Cornell University, Computer Science, July 19, 1999.

[11] Robert Bruce Findler and Matthias Felleisen. Behavioral interface contracts for java. Technical Report TR00-366, Department of Computer Science, Rice University, August 2000.

[12] Robert Bruce Findler, Mario Latendresse, and Matthias Felleisen. Object-oriented programming languages need well-founded contracts. Technical Report TR01-372, Department of Computer Science, Rice University, 6100 South Main Stree, Houston, Texas, 77005, 2001.

[13] Microsoft Research Foundations of Software Engineering, 2001. `http://research.microsoft.com/fse`.

[14] Uwe Glässer, Yuri Gurevich, and Margus Veanes. Universal plug and play machine models. Technical Report MSR-TR-2001-59, Microsoft Research, June 2001. Available from `http://research.microsoft.com/pubs/`.

[15] Wolfgang Grieskamp, Yuri Gurevich, Wolfram Schulte, and Margus Veanes. Testing with Abstract State Machines. In *Formal Methods and Tools for Computer Science, Eurocast 2001*, pages 257–261. IUCTC Universidad de Las Palmas de Gran Canaria, February 2001. Submitted for inclusion in LNCS ASM 2001.

[16] Y. Gurevich. Evolving Algebras 1993: Lipari Guide. In E. Börger, editor, *Specification and Validation Methods*, pages 9–36. Oxford University Press, 1995.

[17] Angel Herranz-Nieva and Juan Jose Moreno-Navarro. Generation of and debugging with logical pre and post-conditions. http://lml.ls.fi.upm.es/slam/.

[18] H.B. Jonker. Ispec: Towards practical and sound interface specifications. In *IFM'2000*, volume 1954 of *LNCS*, pages 116–135, Berlin, Germany, November 1999. Springer-Verlag.

[19] G. T. Leavens and M. Sitaraman (eds.). *Foundations*

of Component-Based Systems. Cambridge University Press, New York, NY, 2000.

[20] Gary T. Leavens and Krishna Kishore Dhara. Concepts of behavioral subtyping and a sketch of their extension to component-based systems. In Gary T. Leavens and Murali Sitaraman, editors, *Foundations of Component-Based Systems*, chapter 6, pages 113–135. Cambridge University Press, 2000.

[21] K. Rustan M. Leino. Applications of extended static checking. In Patrick Cousot, editor, *Static Analysis: 8th International Symposium (SAS'01)*, Lecture Notes in Computer Science, pages 185–193. Springer, July 2001.

[22] Barbara Liskov and Jeannette Wing. A behavioral notion of subtyping. *ACM Transactions on Programming Languages and Systems*, 16(6):1811–1841, November 1994.

[23] Bertrand Meyer. *Eiffel: The Language*. Object-Oriented Series. Prentice Hall, New York, NY, 1992.

[24] Bertrand Meyer. *Object-oriented Software Construction*. Prentice Hall, New York, NY, second edition, 1997.

[25] P. Müller and A. Poetzsch-Heffter. Modular specification and verification techniques for object-oriented software components. In *Foundations of Component-Based Systems* [19], pages 137–160.

[26] Clemens Szyperski. *Component Software*. Addison-Wesley Publishing Company, Reading, Massachusetts, 1999.