Serious Specification for Composing Components

Mike Barnett Wolfgang Grieskamp Clemens Kerer [*] Wolfram Schulte
Clemens Szyperski Nikolai Tillmann Arthur Watson

Microsoft Corporation
One Microsoft Way
Redmond, Washington 98052-6399
USA
E-mail: {mbarnett,wrwg,schulte,cszypers,nikolait,arthurw}@microsoft.com

Abstract

We discuss the use of an industrial-strength specification language to specify component-level contracts for a product group within Microsoft. We outline how the specification language evolved to meet the needs of the component-based approach followed by that group. The specification language, AsmL, is executable which allows for testing to be done using runtime verification. Runtime verification dynamically monitors the behavior of a component to ensure that it conforms to its specification.

1. Introduction

Component technology promises the creation of cheaper, more specialized, and naturally extensible applications [17]. Exploring just that is a new product group that has grown out of the Component Applications group at Microsoft Research [4]. The product group's initiative is still in its infancy and it is too early to report on particular results or experiences of the product per se; instead this paper is about the development process used by the group.

It has become clear that flexible composition requires a strong handle on specification. A composite assembled from a number of components can meet its requirements only if the constituting components meet theirs. Traditional development approaches escape to some degree by relying on integration testing to ensure that the whole meets requirements — in the extreme case allowing for the exact requirements on its parts not even to be known.

This paper is about the exploration, evolution, and application of AsmL [7] to enable precise specifications in the context of a componentization approach on top of the .NET Common Language Runtime (CLR). The componentization approach draws heavily on two concepts: the support for multiple interface inheritance in the CLR and the notion of property setters and getters to establish connections. The approach is similar to the established model found in Java and $C^\sharp$ that suggests connecting event listeners to event sources. However, rather than focusing on events or the connections at the level of individual methods (for instance, using $C^\sharp$ delegates), the new approach focuses on using connections typed by interfaces for all instances of functional dependency.

Using role-based modeling, small well-factored interfaces are defined first. The architecture for particular subsystems is described mostly in terms of such interfaces and how they relate to define entity and relationship types. The first step is to specify interfaces in terms of abstract specification models. In a second step, a combining interface — an interface that itself is empty, but that derives from a set of interfaces — can be defined. The combining interface, in its specification, introduces invariants that couple the separate specification models of the interfaces it combines. This way, separate roles are grouped and ultimately refined to a type that is the basis for implementations (classes implementing such combining types).

A significant design decision in the component model is to use property getters/setters (as, for example, supported directly in $C^\sharp$) to express connection points. Getters and setters are methods that syntactically appear as fields; they provide a

[*]Clemens Kerer participated in this work while he was a summer intern at Microsoft; he is currently a graduate student at the Technical University of Vienna. His email address is C.Kerer@infosys.tuwien.ac.at.

systematic means for attaching arbitrary computations to what are nominally variable references. Without language support, programmers tend to use ad-hoc naming conventions. A connector interface is an interface that offers getters and setters. Getters are used to “pull” sub-objects that implement a particular type — usually one of the combining interface types introduced above. That is, a getter realizes a *provides* interface. Setters are used to connect an object to another object of appropriate type — setters realize *requires* interfaces [17].

In the absence of a firm specification approach, this composition approach is doomed to be useful only in the context of, ultimately, monolithic application construction. No assurance would exist that the combinatorial explosion of possible compositions would all yield desired results. The product team is therefore committed to developing formal specifications for all public interfaces, and testing all components against those specifications.

The rest of this paper is organized as follows. In Section 2, we present a very short description of AsmL and its use for runtime verification. Section 3 describes the development tool chain and presents a simple example. We discuss several issues and a preliminary evaluation in Section 4. Section 5 describes related work; Section 6 describes the future steps that we currently envision for the project.

2. AsmL and Runtime Verification

AsmL is an executable specification language created by the Foundations of Software Engineering group [7]. It is based on the theory of Abstract State Machines [9, 10]. It is also a full .NET language; AsmL programs can inter-operate with any other .NET component, e.g., written in $C^\sharp$, VB, or C++. A specification written in AsmL is an operational semantics expressed at an arbitrary level of abstraction. AsmL incorporates the following features:

Nondeterminism AsmL provides a carefully chosen set of constructs with which one can express nondeterminism. They allow the specification of a range of behaviors within which the implementation must remain. Overspecification can be avoided without sacrificing precision.

Transactions AsmL is inherently parallel: all assignment statements are evaluated in the same state and all of the generated updates are committed in one atomic transaction. Updates that must be made sequentially are organized into *steps*; the updates in one step are visible in following steps and steps can also be organized hierarchically. Removing unnecessary sequentialization also helps prevent overspecification. The specifier describes how the state of the component should be in the next state without necessarily describing the implementation-level decisions that must be made on how to effect the changes. In this paper, a method body is treated as a single step from the specification’s viewpoint.

On top of these basic features of the language, in order to cope with the problems of component-oriented programming, we introduce the notion of *mandatory calls*. A mandatory call is an externally visible communication that a component must engage in, e.g., performing a callback or outcall under certain circumstances.

The executability of AsmL provides an interesting possibility for verification [2]. Instead of restricting the systems or the specification language in order to allow for static verification, we can use an AsmL specification to monitor the behavior of a component. Conceptually, we run the specification and the implementation in parallel and check that the behavior of the latter is a possible behavior of the former. *Runtime verification* functions as a test oracle; this increases the efficacy of testing. The more thoroughly the AsmL specification captures the intended software behavior, the more the testing balance shifts from behavior verification to test case generation, and the more benefit can be derived from techniques such as using random test case generation to improve test thoroughness relative to structural or data flow adequacy criteria.

Where the product group used tools to determine and optimize *tested code coverage* in the past [18], AsmL tooling enables to tackle the equally important area of *tested specification coverage*.

An AsmL specification for a component is described as a *rich interface* [1]. Syntax-only interfaces (i.e., interfaces as they exist in languages such as C♯, C++, or Java) are enriched with fields and method bodies. The fields are of two types: *model variables* which are self-contained in the specification or *abstraction variables* that are used to link the state space of the implementation to that of the specification. Method bodies can be expressed as pre-/post-condition pairs, i.e., declarative specifications, or as *model programs*, i.e., operational specifications, or as a combination of the two.

A model program performs updates on the model variables and performs component interactions in order to effect the state changes — on the level of abstraction of the specification — that should correspond to state changes in the real implementation, if the implementation is correct.

Abstraction variables provide a link between the implementation and the specification. An *abstraction function* is a function which, from accessing the internal state of the implementation, creates the values of the abstraction variables.

Return values can be thought of as abstraction variables that do not need an abstraction function, since the value is made publically available by the implementation.

In the current project, all specifications are fully declarative. Since there are no model programs, any conditions beyond those involving the result returned by a method require an abstraction function to be defined.

Rich interfaces provide for the flexible specification of component composition. Behavioral sub-typing [13] is expressed by invariants and constraints in interfaces that derive from super-type interfaces. Behavioral linkage and aggregation are also easily specified.

3. The Development Process

The central pieces of the development and test architecture concerning specifications are the AsmL compiler, the Weaver, and the Test Execution Framework (TEF). The AsmL compiler, as every .NET compiler, produces IL, the intermediate language of the .NET runtime. The Weaver takes the resulting IL, along with the IL for the implementation, and injects the specification conditions and abstraction objects into the appropriate places in the implementation. This approach of merging IL has significant benefits compared with previous prototype work done by the product team, in terms of both efficiency and the programming model. The previous work used context boundaries to test pre- and post-conditions and invariants by intercepting calls into and out of components under test, and used a combination of component wiring and .NET reflection to provide access to private component implementation state in abstraction implementations.

The TEF also interacts with other sub-systems for test case generation, test result persistence, and a test case manager, but those are not relevant to this discussion. We discuss the details by following the work flow as it relates to specification.

The starting point for writing a component is the set of rich interfaces that the component is supposed to meet. We consider only purely declarative specifications in this section, i.e., pre- and post-conditions and invariants for an interface. (A refinement that introduces model programs is possible at any time.) The conditions are defined over a set of abstraction variables. Due to space considerations, we present a synthetic example that contains the important features.

```
interface Sample
  [ByAbstraction]
  var x as Integer
  constraint x mod 2 = 0
  f(i as Integer)
    require i mod 2 = 0
    ensure resulting x = x + i
```

This example defines an interface with a single method f that takes an integer as a parameter. The integer field, x, is not part of the “native” interface that an implementation must include, but is used to express the conditions in the specification. The rich interface also contains an invariant: that x will always be an even number. Again, no implementation needs to have a corresponding variable; a correct implementation is one that would never violate the invariant.

The pre-condition for the operation f is that the argument must be an even integer; the post-condition states that the final value of x should be its initial value incremented by i (and thus always even, since it must be even at creation in order to satisfy the invariant). The keyword *resulting* is used to refer to the final value of a variable; the variable by itself refers to its initial value even if it occurs in the post-condition. The marking *ByAbstraction* is a *custom attribute*; this is a feature of .NET that allows meta-data to be attached to elements of a program (and, thus, to an AsmL specification). This particular custom attribute means that x is an abstraction variable (its value is to be retrieved “by abstraction” from the implementation).

Note that it appears that the invariant could have just been added to the conditions on f. The difference is that the invariant is binding on any interface which extends this one; i.e., all sub-types must be behavioral sub-types. Given such an interface, the AsmL compiler generates a *contract class*, a *native interface*, and an *abstraction interface*.

Contract Class A contract is a class that is in a particular format which the Weaver uses as input to inject into the implementation. It contains a method for the class invariant and a method for each pre-condition and post-condition.

For the example, there would be a method for the invariant, a method f_pre, and a method f_post. In addition, the field x would exist.

Native Interface The native interface contains only the signatures of all the public methods (and properties, etc.) that are understood by other programming languages, in particular the language that the implementation is going to be written in. When the implementation class is written, it will list the native interface as one of the interfaces it supports. If the rich interface specifies an interface that already exists, i.e., as part of an a priori given library, then the native interface that is generated by the AsmL compiler is checked by the Weaver to be identical to the already existing interface and ignored subsequently.

For the example, the native interface (in $C^\sharp$ notation) would be:

```
interface Sample {
  void f(int i);
}
```

The interface is CLR-compliant; it can be implemented by a class in any .NET language.

Abstraction Interface The abstraction interface contains a property for each abstraction variable. The property is read-only and is used by the methods in the contract class to retrieve the (abstracted) value of the implementation's state.

For the example, the abstraction interface would be (again, using $C^\sharp$ notation):

```
interface Sample_Abstraction {
  int x { get; }
}
```

Before testing can be done using runtime verification, an *abstraction implementation* class must be created for each implementation class under test. The abstraction implementation class implements the abstraction interfaces corresponding to each interface supported by the implementation class.

Although this class is compiled separately from the implementation class, it has full access to private fields of the implementation class (if the appropriate security settings allow): the abstraction class redefines any private fields that are needed (with the proper names and types) and these "shadow" fields are then removed by the Weaver during the IL merging process. Of course, the developer of the abstraction implementation must inspect and understand the implementation class. The important point is that the computational dynamics of the implementation do not need to be understood, but only how its data structures correspond to those of the specification. Since the types of the abstraction variables are AsmL datatypes, it is particularly convenient to use AsmL to write the abstraction class.

Now we assume that the implementation of the component has been accomplished and that the abstraction class has been written. To finish preparing the component for testing, the TEF locates any specifications for the components involved in the test (potentially one for each interface that the components implement) and the corresponding abstraction classes. These two inputs are provided to the Weaver along with the actual implementation. Note that all three are in IL since each has been generated by a .NET compiler. All three can be written in different source languages, although, as we noted earlier, the use of AsmL data structures in the specification make it most likely that the abstraction class will have been written in AsmL as well.

The Weaver then combines the IL for the three inputs into one unified binary. All of the methods and fields of the contract class are added to the implementation class (except for any shadow fields). The methods of the implementation class are re-written so that, e.g., f calls f_pre at the beginning of the method and f_post at the end of the method. We currently check invariants both at the end of every method and at the beginning of all non-constructor methods. Full mandatory call support is not yet implemented, but invariant checks will also occur at those points; the idea is that all invariants should be in force whenever a client could observe the state of a component. The original body is wrapped in a try-finally block so control cannot be inadvertantly returned without all of the appropriate conditions being checked. The abstraction class also has all of its fields and methods moved over into the implementation class. This way, no extra wiring is needed: when any abstraction variables are evaluated, the abstraction properties are executed internally and transform the original implementation state into the specification state.

As a result, when executed, each method performs all of the actions of the implementation, but each method's specification is checked at the appropriate moments. If the pre-state is needed in a post-condition, then a copy is made of its initial value so that it is available in the post-condition. (In reality, it is more complicated, but this description suffices for the simple cases.)

At this point, testing proceeds just as it would for an un-instrumented implementation, either testing components in isolation through the TEF or at any level of integration up to a full application context. The difference is that any violations of the conditions in the specification result in a particular type of exception being thrown.

The product team developed a number of small utilities, integrated into the Microsoft Visual Studio .NET development environment, to streamline the mechanics of the specification and testing process. One utility creates a new rich interface template based on an existing interface and adds custom attributes so the Weaver can associate the two. Another utiltity creates an abstraction implementation template based on an existing class. A final utility creates a test template based on an existing interface. These utilities also manage the integration of the generated code with the source code control and build systems. As a result, the product team's efforts are focused on the intellectual content of interface specification and testing.

4. Discussion

Transferring specification technology to a product group is not a simple process. We have found that one cannot simply "hand it over" and expect to see any lasting change to the development process. Instead, the two teams have had to engage in a lengthy adoption process during which several interesting developments have occurred. It is important to stress that the project is still at an extremely early stage.

4.1. Declarative vs. Operational

One effect of the project has been feedback on the specification language itself. AsmL has always been more oriented towards a "model program" view of the world, i.e., a specification expresses, via its operations on the abstract state, how a corresponding implementation should act. The product group, however, has a historical attachment to specifications expressed as pre- and post-conditions.

We believe there is a duality between a declarative specification and a model program. A declarative specification encodes in a static fashion the state change that is produced by executing a model program. AsmL allows the use of either; however declarative specifications cannot be executed in isolation. There are additional advantages of executing a specification in a stand-alone mode, such as test-case generation [8] or use-case validation.

But as a result of this project, the support for declarative conditions coupled with abstraction objects in AsmL has been increased.

4.2. Influences on the Development Process

Another effect of the project has been the design feedback offered by the necessity of describing the semantic consequences of design decisions. The need to encode the meaning of operations within interfaces forced a reconsideration of the interface factorization.

It is important to note that the modified AsmL and supporting tool chain were applied to larger examples taken from the componentization project. In particular, a finely factored Tree API supporting the manipulation of XML-like virtualized data structures was used as an experimental subject. The API was the result of a prototyping activity and consists of about twenty finely factored roles that are then combined into various types.

AsmL as it stands at the end of this project appears to be rich enough to cover delicate cases, such as the Tree API. It has already delivered real value in uncovering factoring faults of the original Tree API design. For instance, the intuitive difference between two different abstraction levels (two different combining types) wasn't confirmed: at a precise specification level, both turned out to be identical. In other cases it was found that, at a specification model level, previously factored roles were so tightly related as to suggest integrating them into a single role/interface type.

This is exactly the kind of result that we hoped for: the earlier these types of design decisions can be resolved, the fewer costly changes that will need to be made later on. It is often too late to fix them when code has been written that has "baked in" unfortunate choices.

5. Related Work

While there is a vast literature on specification and verification, we limit this section to the work most closely related to runtime verification. Perhaps the closest work is the JML runtime assertion checking provided for components written in Java [12]. Eiffel [15] also provides for the checking of pre- and post-conditions, but only for components written in Eiffel. There are many similar design-by-contract tools for Java, such as JMSAssert [14], iContract [11], Handshake [5], Jass [3], and JContract [16]. However, all lack any facility for maintaining the state-space separation between the specification and the implementation. More general component-oriented work has been done by Edwards [6] to generate wrapper components for checking pre- and post-conditions, but does not seem to handle more general synchronization issues that require model programs. In general, all of these systems impose a large set of restrictions on what can be expressed in their specification languages.

6. Future Work

There are several issues that we already foresee. Our current framework provides notification that a component does not meet its specification, but there is much automated support that could help pinpoint the reason for the failure.

It remains to be seen whether the developers will be willing to define all of their interfaces as rich interfaces and how meaningful the conditions will be. The hope is that such effort will pay for itself by reducing expensive design and coding mistakes and by improving test efficiency and effectiveness.

We are also interested in tracking whether the specifications will remain purely declarative, or whether they will begin to be expressed as model programs. As component interaction issues are specified, this will become increasingly important.

Acknowledgements

This work would not have been possible without the efforts of the entire Foundations of Software Engineering group at Microsoft.

References

[1] M. Barnett and W. Schulte. The ABCs of specification: AsmL, behavior, and components. *Informatica*, 25(4):517–526, Nov. 2001.

[2] M. Barnett and W. Schulte. Runtime verification of .NET contracts. *The Journal of Systems and Software*, 2002. To Appear.

[3] D. Bartetzko, C. Fischer, M. Möller, and H. Wehrheim. Jass - Java with assertions. In K. Havelund and G. Rosu, editors, *Proceedings of the First Workshop on Runtime Verification (RV'01)*, volume 55 of Electronic Notes in Theoretical Computer Science. Elsevier Science, July 2001.

[4] M. R. Component Applications, 2002. `http://research.microsoft.com/comapps`.

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

[6] S. H. Edwards. A framework for practical, automated black-box testing of component-based software. *Software Testing, Verification and Reliability*, 11(2), 2001.

[7] M. R. Foundations of Software Engineering, 2002. `http://research.microsoft.com/fse`.

[8] W. Grieskamp, Y. Gurevich, W. Schulte, and M. Veanes. Generating finite state machines from abstract state machines. *Software Engineering Notes*, 27(4):112–122, 2002. From the conference International Symposium on Software Testing and Analysis (ISSTA) 2002.

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

[10] Y. Gurevich. Sequential Abstract State Machines Capture Sequential Algorithms. *ACM Transactions on Computational Logic*, 1(1):77–111, July 2000.

[11] R. Kramer. iContract—the Java Designs by Contract tool. In *Proc. Technology of Object-Oriented Languages and Systems, TOOLS 26, Santa Barbara/CA, USA*. IEEE CS Press, Los Alamitos, 1998.

[12] G. T. Leavens, Y. Cheon, C. Clifton, C. Ruby, and D. R. Cok. How the design of JML accommodates both runtime assertion checking and formal verification. Technical Report 03-04, Department of Computer Science, Iowa State University, Ames, Iowa, 50011, Mar. 2003. To appear in the proceedings of FMCO 2002.

[13] B. Liskov and J. Wing. A behavioral notion of subtyping. *ACM Trans. Prog. Lang. Syst.*, 16(6):1811–1841, Nov. 1994.

[14] Man Machine Systems. JMSAssert, 2002. Available from `http://www.mmsindia.com/JMSAssert.html`.

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

[16] Parasoft Corporation. Using design by contract to automate java software and component testing, 2003. Available from `http://www.parasoft.com/jsp/products/article.jsp?articleId=402`".

[17] C. Szyperski, D. Gruntz, and S. Murer. *Component Software: Beyond Object-Oriented Programming*. ACM Press and Addison-Wesley, New York, NY, 2nd edition, 2002.

[18] C. Wittenberg. Components in the key of C. In *OOPSLA*, 1998. research.microsoft.com/comapps/docs/oopsla98.ppt.