Component Technology - What, Where, and How?

Clemens Szyperski
Microsoft Corporation
research.microsoft.com/~cszypers/

Abstract

Software components, if used properly, offer many software engineering benefits. Yet, they also pose many original challenges starting from quality assurance and ranging to architectural embedding and composability. In addition, the recent movement towards services, as well as the established world of objects, causes many to wonder what purpose components might have.

This extended abstract summarizes the main points of my Frontiers of Software Practice (FOSP) talk at ICSE 2003. The topics covered aim to offer an end-to-end overview of what role components should play, where they should be used, and how this can be achieved. Some key open problems are also pointed out.

1. Why components?

1.1. Words ...

Composition is the act of applying a composition operator (that forms part of a composition model and theory) in a given context. *Components* are the subjects of composition. *Composites* (also called *assemblies*) are the results of composition. This about as much as can be said if we stay at the most abstract level.

To move to meaningful discussion and elaboration, we need to focus in on a particular domain of composition, leading to families of composition models and theories. In this paper, our focus is component software and components are thus *software components* [1,2]. To be sure: there are many component domains in the larger space of software engineering and software architecture. Most of them are not software components, in the sense used here, but that doesn't affect their value – merely their scope of applicability.

Hence, instead of listing technically motivated criteria that capture our domain of discourse, let's explore the ultimate reasons for what software components are meant to achieve. Then, we can work our way backwards and establish useful technical criteria.

1.2. Why components?

There are at least four 'tiers' of motivations for using software components. The oldest tier is along the lines of *Make and Buy*, grounded in the observation that many if not most organizations need to own some edge on top of acquired baseline products. Essentially, organizations need to strike a balance between the promised flexibility and competitive advantage of purpose-built software and the economic advantage of standard software. This is the space of traditional software reuse thinking and, often, it suffices to focus at the level of source components. *Source components* are components that are consumed at software development time and include architectural, design, and source code artifacts.

The second tier follows the line of reusing partial design and implementation fragments across multiple solutions or products. This is the space of software product lines and product line engineering. Often, it suffices to focus on *build-time components*, components that are consumed when building deliverable software out of multiple pieces. Like source components, build-time components do not

0-7695-1877-X/03 $17.00 © 2003 IEEE

necessarily survive the build process as identifiable parts of the deliverable. Traditional linkers are good examples of build-time tools that fuse their input into a single output.

The third tier assumes that components from multiple sources are integrated on site, that is, not as part of the software build process. Such customer-driven integration is typically called *deployment* and the matching components can be called *deployable components*. Deployable components are 'real' software components in the sense that they are units of deployment that, as such, remain identifiable. Deployment is usually distinguished from installation as being the last step allowing for detailed configuration or customization. Once deployed, a component can then be installed on many systems. A good example in this category is a web browser incorporating downloaded components into the functionality of an active web page. While the browser performs automatic installation, it is the web developer that deployed the component (and thus had an opportunity to customize and test it in the context of the particular web site).

The fourth tier is concerned with dynamic servicing, upgrading, extension, and integration of deployed systems. Varying degrees of redeployment and automatic install and uninstall serve this space. The desire grows to engineer solutions that – over their lifetime time – can deal with new and evolving contents, schemas, and services. This leads towards requirements to enter tier four; a tier that is mostly the realm of ongoing research.

1.3. Dynamic Upgrade and Extension

The most refined tier of applying software components, that of dynamic upgrading, extending, and integrating in running systems, is much at the cutting edge of current technology and understanding. Practical use of components today tends to end with the third tier: deployment of components. The most prominent example in this space are application servers (mostly J2EE/EJB and .NET/COM+).

Truly dynamic changes to solutions build on components are extremely challenging in terms of correctness, robustness, and efficiency of the resulting systems. However, even at tier three, most of the problems already surface. It cannot be assumed that an organization that deploys components is capable or willing to perform deep end-to-end integration tests. Even where such tests are performed (because, say, the solution in question is mission critical), it cannot be assumed that the deploying organization can do anything but outright reject the use of a third-party component that has serious problems. In particular, it cannot be expected that such components will be 'fixed' before deployment proceeds.

These observations lead to a quality assurance issue at the component supplier end: since components can be combined into an endless variety of compositions, there is no opportunity to perform any final integration tests at the component supplier's end. Component quality needs to be established in the absence of a closed-world assumption! In other words, component unit tests and other forms of quality assurance, such as verification of component properties, are of critical importance.

Truly dynamic contents and truly open sets of services cannot be handled by closed software. The resulting need to support tier four components grows as standards in the space of XML and web services lead to the broad availability of open contents and dynamically located service functionality. To cope with such situations, systems need to on-demand locate, install, and integrate components – or remotely use the dynamically discovered services.

A strong motivator to move into this technically difficult area is the desire to integrate across organizational boundaries. From enterprise application integration (EAI), to business-to-consumer and business-to-business (B2C and B2B) scenarios, to full peer-to-peer (P2P), it gets ever less likely that homogeneous schemas and protocols can be assumed.

1.4. Component maturity model

The multiple tiers of component concepts explored in the previous section, coupled with the observation that higher tiers require more refined competencies, lead to a simple *component maturity model*. (The familiarity of the resulting acronym is entirely coincidental.) In order of increasing organizational maturity requirements, the following levels can be distinguished:

1. *Maintainability*: modular solutions
2. *Internal reuse*: product lines
3.a.1 *Closed composition*: make and buy from closed pool of organizations
3.a.2 *Open composition*: make and buy from open market
3.b *Dynamic upgrade*
4. *Open and dynamic*

Somewhat depressingly, the state of the art of many software solutions doesn't even embrace modularity. Even advanced organizations are presently concerned with mostly level 2 issues.

Level 3 forks into two parallel options that, in combination, form level 4. The first fork opens solutions for third-party contributions – in two steps from select ('pooled') sources to open markets. The second fork opens solutions for dynamic upgrade. It is the combination of these two forks that leads to the most demanding forms of component use.

1.5. Compositional reasoning

To climb up the maturity ladder towards mastering systems that support open and dynamic composition, effective *compositional reasoning* at all levels is required. Compositional reasoning builds on modular reasoning: the ability to reason within confined scopes without needing to resort to any form of global inspection or analysis. In a proper compositional-reasoning framework, the results from modular reasoning 'survive' application of useful composition operators.

Some will say: 'Hey, it's just mathematics!' However, there is much room for future work. For instance, predictable assembly is a focus of current research: how can important assembly properties be predicted reliably, assuming just the known properties of components used and the known inference rules of a particular composition theory. Assembly properties of interest span the range of both functional and extra-functional properties.

2. What's a component?

2.1. What's a Software Component anyway?

The analysis of what it is that we would want from software components (sketched in the previous section), leads to a first approximation of what such components have to be. In particular, a software component has to be a *unit of deployment*. Furthermore, to enable dynamic scenarios, it has to also be a *unit of versioning and replacement*.

To be a unit of deployment, a software component has to be an executable deliverable for a (virtual) machine. To be machine executable, no human intervention should be required to turn the combination of a deployable component and a deployment descriptor into an installable component, ready for execution. This is because deployment is not a development activity and does not happen at the component supplier's site. While the deployment process does provide extra information (captured in a deployment descriptor), it shouldn't require a build environment or the presence of a developer.

To be a unit of replacement and versioning, it is important that a deployed software component remains invariant as it gets installed onto possibly many systems. That is, installed components should carry any *no observable state*. Therefore, software components live at the level of packages, modules, or classes, and not at the level of objects or distributed objects.

To fully explore the space of code and data, in their co-packaging in a component, it is useful to view software components as a collection of modules and resources. Modules contain immutable code (for instance, in the form of a set of classes). Resources contain immutable data (for instance, in the form of serialized objects). Immutable metadata can be contained in both modules (describing code) and in resources (describing data). In a deployed component (that is, an installable component), deployment information may have been folded into resources, modules, or both. For instance, some application server implementations meet requirements found in deployment descriptors by generating modified code from the code found in a deployable component.

2.2. What is Deployment?

Acquisition is the process of obtaining a software component; as discussed above, such a component arrives in deployable form. *Deployment* is the process of readying such a component for installation in a specific environment. The degrees of deployment freedom are typically captured in *deployment*

descriptors, where deployment corresponds to filling in parameters of a deployment descriptor.

Installation is the process that follows deployment and that is often automated. Installation makes a component available on a particular host in a particular environment.

Loading is the process of enabling an installed component in a particular runtime context, such as a process. If a component carries definitions for 'static' (or 'global') variables, then these are instantiated as a side effect of loading. Finer-grained *instantiation* in the object-oriented sense follows loading and operates over parts of a loaded component (typically classes). Where instances have persistent state, such state is not carried by a software component, but needs to be mapped into a specific external store.

3. Components in Context

A software component:

- assumes architectural embedding;
- presents functionality via interfaces (these are called 'incoming' or provides-interfaces);
- has parametric dependencies via interfaces (called 'outgoing' or requires-interfaces);
- has static dependencies;
- targets specific component platform;
- requires other components; and
- requires per-instance context.

Of these, provides-interfaces are the traditional focus as they seem to capture the abstraction implemented by a component. Ignoring the others leads to implementations that cannot be composed. For example, classes do not compose in general.

To enable composition (and thus components that deserve their name), all significant dependencies and assumptions of a component's implementation also need to be captured.

Architectural embedding and various forms of dependencies are discussed in more detail in the following subsections. Per instance contexts are not discussed in this paper; this category includes the process hosting an instance and the container (or context) enclosing it.

3.1. Architectural embedding

Specifications (and implementations!) need to be grounded in a framework of common understanding. At the root is a common *ontology*, ensuring agreed upon terminology and domain concepts. Equally important are accepted conventions and best practice. In combination, these allow separate parties to communicate and (importantly) silently share a substantial context of assumptions.

Architecture contributes top-level factorings of quality responsibilities and reference models. Building on such reference models (or reference architecture), components can be designed that target specific niches. The result is that components do not fit into solutions by coincidence but by construction. Problems of component adaptation and impedance mismatch are reduced to exceptional situations. In other words, all good engineering minimizes the need for adapters.

(Note that 'glue code' is misnamed if it includes adaptation logic. Glue, in the real world, works only if applied to surfaces that already match each other closely. Gluing is not an adaptation but a composition technology.)

3.2. Parametric dependencies

To enable the use of components in many compositions, it is important that instance-level dependencies can be configured. A requires-interface is a means to this end: it states what an instance would need to function. In its simplest incarnation, a requires-interface is just the type of an instance variable. Setting that variable to refer to some other instance of appropriate type is then equivalent to *connecting* the two instances. Requires-interfaces lead to *parametric dependencies* because the dependency on the interface type does not imply the use of any particular (complete) implementation.

The concept of requires-interfaces can be generalized to *requires-types*. Exploring suitable type and composition systems is the subject of ongoing research. Examples include certain module systems, units, mixin layers, and parametric contracts.

Parametric dependencies are usually preferred over static component dependencies. However, not all static dependencies can be eliminated (following subsection). In addition, it is surprisingly possible to model parametric dependencies as a special case of

static dependencies. The idea is to treat the type of the requires-interface as a regular type that is packaged as part of the metadata contained in some other component. In the limit case, all static type dependencies can be decoupled from implementation choices by only allowing static dependencies on pure metadata (pure types without any attached implementation).

Components that contain no implementation but only metadata are useful for reasons of reflection, dynamic composition, and so on.

3.3. Static dependencies

Static dependencies fall into three categories: dependencies on the component platform, dependencies on other components, and dependencies on the deployment context.

Dependencies on the *component platform* are ultimately unavoidable since it isn't practical to aim to parameterize everything. Examples of such dependencies include:

- the deployment format; required to even recognize a component as one – think JAR/EAR files or .NET assemblies;
- the (virtual) machine model – including the instruction set;
- mechanisms such as proof-carrying code for various forms of verification;
- ways to access well-known services – addressing the bootstrap problem (consider directory service);
- the supported *component model*: what is a component on this platform? What is the security model? Or that for authentication, trust, or certification?

Dependencies on other components serve many purposes. Ultimately, at least one such dependency per used component must exist somewhere or else that component would not even be loaded. (In highly configurable systems the single dependency might exist in a configuration file. However, in a pure model, that file would itself be the resource part of a component.)

An extreme form of static inter-component dependencies is a traditional Modula-2 or Ada style module system. Every module carries a list of static imports of other modules. There is no compositional freedom – each module is added in exactly one way.

At the other end is an extreme approach of no dependencies on other components at all. This could be called the game machine approach, since gaming machines (like Playstation, GameCube, or XBox) traditionally assume close to no software pre-installed on the machine and everything coming with the game. Such a game can be modeled as one (huge) component that has no dependencies on other components. The result is that no integration is provided above hardware primitives. If the hardware layer opens any communication channel, such as access to the Internet, then any component that carries a network stack can communicate with any other component that does so as well. This is an example of entirely parametric dependencies.

Dependencies on the *deployment context* are a per-instance concept. Each instance created by a component is placed in a context: an operating system process, a COM apartment, a COM+ context, an EJB container, a CLR AppDomains, and so on. Sometimes, deployment contexts are 'woven' into a component at (or before) deployment time or into instances. This corresponds to techniques used for static and dynamic aspect-oriented programming, respectively.

3.4. Atomic vs. composite components

Atomic components are created 'from first principles'. To make composition models generally useful, they need to provide for hierarchical composition means. Indeed, by simply treating composites as components again, hierarchical composition is a natural outcome.

A composite references the components over which it composes and typically adds modules or resources (or both) of its own that encapsulate the actual composition operator applications. That is, a module in a composite might contain the 'glue code' that instantiates and connects the components. Alternatively, a composite's resources could contain a serialized graph of instances that represent a composition prototype. Instantiating a composite amounts to executing its glue code or deserializing its prototype instance graph. (Since a composite is still just a collection of modules and resources, no extension of the general component characterization above is needed.)

It is important to understand that composites do *not* contain the components over which they compose. That is, all components exist in a flat universe. This is an important property, as it allows servicing of components without having to know all places where that component has been used. (A practically important example of servicing is the closing of a security hole.)

4. Naming, versioning, side-by-side

The technical process leading to software components is complex. The total number of components that may coexist in a system is huge. Component developers need to be able to name components without risking collisions with other developers. Components developers also need to be able to release new versions of their components; especially if they have been successful. Component deployers need to rely on strong component naming to ensure that the right components come into play. Component deployers also need to be able to affect versioning resolution to maintain robust configurations.

4.1. Making a component

The making of a component involves a large number of different inputs and typically involves a complex process and likely many individuals. (The latter is not true for fine-grained components. However, the former remains as fine-grained components are likely build in large sets, requiring significant build systems for economies of scale.)

Inputs into the build system delivering a deployable software component include:

- The collection of sources;
- all tools used, including compilers, transformers, generators, optimizers, linkers;
- referenced artifacts, such as other components listed as static dependencies;
- the build system itself, including the settings in the build environment.

4.2. Component versioning

Every input into the build process potentially affects the built component. That is obviously and intentionally the case for the sources. The impact of the build environment is easily underestimated. Only very careful build system setups or methods reduce this impact. For instance, search paths are a wide-spread evil. Referenced artifacts are expected to have an impact at the level of exposed metadata, but unless care is used, unexpected dependencies can sneak in. An example is the unintended propagation of compiler optimizations, such as offset calculations.

While it is generally understood that all involved tools have an impact, it is difficult to fully account for these dependencies. For instance, upgrading a compiler to a newer version can easily lead to different components, even if everything else is unchanged and even if the old and new compiler versions are provably equivalent. (The new component might have different performance or footprint characteristics.)

Thus, unless the resulting bits are one-for-one identical, it has to be assumed that a new version has been produced. This is a conservative rule, but moving to a more precise one requires great semantic care: Functional and extra-functional refinement need to be demonstrated, which is definitely hard in the general case.

4.3. Versioning and side-by-side

Version changes have transitive impact. Consider a component A that the makers of a component B used in its version 1. The makers of a component C used A as well, but in its version 2. A problem arises if another party is interested in using B and C in the same context. That is, if that latter party wishes to build a component D that depends on B and C, then loading D requires transitive loading of A – but in two different versions.

There are two fundamental strategies: either A is loaded only once (forcing B to roll forward to version 2 of A) or each version of A is loaded (forcing A and all its clients to tolerate the side-by-side existence of two versions in the same context).

In order to treat this situation soundly, components need to ship with an unambiguous name, which needs to include their version designation. Also, dependencies on other components need to be recorded explicitly, completely, and in a version-precise way. Finally, a component needs to indicate whether it can be loaded side-by-side in multiple

versions or whether it should be rolled forward to the latest version available.

Then, as a design discipline, side-by-side components need to be factored from non-side-by-side ones. Deployed configurations need to allow for the coexistence of side-by-sideable versions and need to bind to the correct versions.

4.4. Side-by-side challenges

Fully supporting side-by-side installation of the majority of components promises the greatest degree of isolation and thus system robustness in the presence of component versioning. At the other end, in-place upgrading of components guarantees the occurrence of the phenomenon known as 'DLL hell'.

The problem with side-by-side support is its interference with cross-component integration. The degree of coupling between any two components determines how feasible side-by-side installations are. The most lightweight form of coupling is through shared pure types and contracts, the most heavyweight through dependencies on implementation detail.

To enable side-by-side integration for at least the most lightweight coupling, it is important to support at least the side-by-side existence of type versions and the simultaneous use of multiple versions of a type within a single component. For instance, if a class implements an interface in multiple versions, then the methods on those interfaces should not be 'folded' into a single implementation – a common mechanism in several popular object-oriented languages.

Enabling proper side-by-side coupling through other dependencies is increasingly harder. The case of side-by-side implementation inheritance is known to the surgeon general to lead to serious health risks.

5. What's a service?

Software services, especially in the specific shape of XML web services, are promising new levels of software integration and interoperability. Understanding how they relate to software components is critically important to benefit from the distinct properties of services without losing the separate advantages of components.

To put it simply, a service is an instantiated configured system that is run by a providing organization. That is, a service is fully grounded. Ultimately, it includes the power supply to the server machines as well as the organization that somehow manages to pay the power bill.

The service-providing organization installs, runs, maintains, and evolves hardware and software infrastructure and components. It provides physical and organizational means, including functions like client management, accounting, and so on.

5.1. Service-level agreements

Since a service is fully grounded and backed by a provider, it can be held to the standards of a service-level agreement (SLA) or a service contract. For instance, a service client signing such a contract with a provider might pay for the service, while the provider guarantees properties such as minimal up-time, performance, or capacity.

It is possible to abstract from the service instance to a suitable service type. At that level, services behave much like objects and service types like classes. External composition of services is just as limited as it is for objects and classes. However, the granularity is very different: viable services are much heavier than typical objects. Therefore, there is a trade-off between the strong guarantees and the limited composability that a service can offer.

Component properties, in contrast, are captured in technical contract between component and client implementers. Such contracts cannot offer service levels, but can (and should) offer parametric means to establish service levels when using a component to build a service. For example, while a component cannot guarantee performance or up-time, it can express its performance or redundancy needs relative to its parametric dependencies.

5.2. Contracts galore

Services, through the possible service-level agreements, offer per-use value in ways that components do not. It is thus justified (and common practice in other industries) to charge for services on a per-use or subscription basis. Such income offsets the real cost of providing a service, leading to plausible business models.

The offering of services under service-level agreements cannot be performed with absolute reliability. This is a standard aspect of any business

and established mechanisms such as auditing and insurance can be used to mitigate the business risk.

The corresponding mechanisms in the component space are very different: components can be verified and certified to meet their specifications (contracts). In the ideal case of total verification, this is a once-and-for-all activity. Once published, a component remains an immutable artifact. Auditing a component, for example, is only meaningful when deployed into a full system, which makes it only a part of the larger activity of auditing a service.

6. Component specification and test

Compositions tend to exhibit a weakest link phenomenon: many compositions are as strong as the weakest component they compose over. Composites can be made stronger than that by applying error containing and handling mechanisms that ultimately rely on various forms of redundancy.

Even in the presence of such measures it is usually effective to aim for a higher reliability of the used components. Reliability of software is a curious notion: in a sense, a correct component is 100% reliable and a component that has the slightest defect is actually incorrect and thus 100% unreliable. In practice, defects do not show in all configurations and not under all load profiles. Therefore, reliability becomes an interesting measure, one asking for a careful foundation, though.

Instead of pursuing this thought further, the following subsections focus on ways to get components closer to correctness.

6.1. Component contracts

As explained earlier, interfaces play a crucial role in any world of components. Assuming that each interface has an attached specification (contract), there are two correlations over sets of interfaces that can be considered. Both turn out to be important in practice, though most approaches (including the one discussed in the following subsections) only focus on the former.

The first correlation is among all requires and provides-interfaces of a single component. One way to look at such a correlation is to view it as a set of invariants that couple model variables (specification variables) of the involved interfaces. For example, consider a component that has a provides-interface that delivers values that are the result of applying a transformation over values acquired through a requires-interface. The per-interface contract of the provides-interface should not mention the requires-interface (or its contract). Instead, it will contain a model variable referring to the abstract stream of incoming values. Likewise, the requires-interface introduces a model variable for the abstract stream of received values. The component-level invariant correlates the two by stating that these two model variables are always of equal value.

The second correlation is sort of dual to the first: it considers interfaces occurring on multiple components and how they need to be correlated to enable useful protocols. The only example known to the author of such inter-component interface correlations are Hans Jonkers' interaction specifications.

Such fine factoring of interfaces and their specifications is itself an interesting challenge. One way to even achieve at interfaces rather than classes is to use role-based modeling. That is, instead of following the path of entity-based or object-oriented modeling, roles are identified and fleshed out. Atomic roles lead to interfaces. Combined roles turn into multiply-derived interfaces that derive without adding any new features. The contracts of such combining interfaces are non-trivial as they correlate model variables introduced by the combined interfaces. (In a sense, such combining interfaces represent a third form of interface contract correlation.)

Once finely factored interfaces are identified, entities or objects that combine multiple interfaces can then be introduced in endless variations. However, the questions of how to specify contracts at this level, how to capture, validate, verify, or test remain.

6.2. One concrete approach: AsmL

There are many specification languages and approaches that support componentization to varying degrees. An example is the abstract state-machine language (AsmL) that is based on the theory of abstract state machines (formerly called evolving algebras). AsmL is being developed by the Foundations of Software Engineering (FSE) team at Microsoft Research in Redmond. The language definition, tutorials, and tools are freely available from their site: *http://research.microsoft.com/fse/*.

AsmL is used as an example here because it is now being used by some Microsoft product groups, leading to a practical refinement of the toolset. Nevertheless, AsmL is work in progress. Also, no claim is made that AsmL will ultimately be the one and only such tool and approach.

At the heart of ASM is the intention to capture operational semantics at a level of abstraction natural to the modeled process. Concretely, AsmL supports *executable model classes* and, inspired by the needs of component technology, *rich interfaces*. Originally targeting COM, the AsmL tools have been retargeted and extended such that AsmL is now a first-class CLR-hosted language. Rich interfaces combine all information required to generate full CLR interfaces with model-level specifications. Models can be a combination of declarative specifications (pre- and postconditions, assertions, invariants) and executable specifications (model programs). Atomic transactions, non-determinism, and mathematical types (such as sets, maps and sequences) help preventing overspecification: the single biggest danger when using operational semantics.

The connection between implementations and model specifications is established through abstraction functions. Test harnesses can use these functions to observe implementation state and test that it meets predicted model state. Declarative aspects of the specification can be injected into implementations to run-time check invariants, pre- and postconditions.

The overall emphasis of AsmL is on specification capture, validation, and implementation test, including automated test case generation. Clear opportunities for verification, checking, and correctness by construction (such as formal refinement) are present, but presently not exploited.

6.3. Example

Below is an example of a simple interface (*System.Collections.IEnumerator*, part of the .NET Framework).

```
interface IEnumerator
{
   object Current { get; }
   bool MoveNext ();
   void Reset ();
}
```

One possible informal specification is captured in the following paragraph:

> Initially, the enumerator is positioned before the first element in the collection. *Reset* also brings the enumerator back to this position. At this position, calling *Current* throws an exception. Therefore, you must call *MoveNext* to advance the enumerator to the first element of the collection before reading the value of *Current*. *Current* returns the same object until either *MoveNext* or *Reset* is called. *MoveNext* sets *Current* to the next element.

A formal AsmL specification of the same interface takes the following shape:

```
public interface IEnumerator
   public var visited as Set of Object
   public var unvisited as Set of Object
   public var current as Object
```

Here, *visited* is the set of elements 'handed out', *unvisited* is the rest of the elements, and *current* is a distinguished element. The specification of *Reset* is:

```
public Reset()
   ensure
      resulting current = null
   and
      resulting visited = {}
   and
      resulting unvisited = unvisited + visited
```

The clause *ensure* states a postcondition, where the modifier *resulting* refers to the after state of that variable and where the + operator denotes set union. A possible implementation of *Reset* might look like:

```
public class MyEnumerator:
System.Collections.IEnumerator
{
   private object[] myElements;
   private int myIndex = -1;
   ...
   public void Reset()
   {
      myIndex = 0;        // ERROR: should be -1
   }
   ...
}
```

6.4. Abstraction Functions

An abstraction function is now used to connect the *Reset* implementation to its specification. For example, it establishes that *current* is *null* if *myIndex* has the value -1 and that it has the value stored at *myElements*[*myIndex*] otherwise.

Since the range of abstraction functions covers AsmL types and since AsmL supports the full set of CLS and most of CTS types, it is possible to define abstraction functions in AsmL. (CLS and CTS are the common language specification and the common type system specification accompanying the CLR. The CTS is the union of all CLR-supported type concepts. The CLS is the CTS subset that is used to enable inter-language interoperability.)

Special so-called shadow fields in AsmL specifications grant access to private implementation fields. Here is the abstraction function for the model variable *unvisited*:

```
public property unvisited as Set of Object
  get
   return { myElements(i)
     | i in [myIndex+1..myElements.Length-1] }
```

6.5. Test of Implementation

Unit tests are performed simply by using the test subject in a number of test cases. There is no need to write a test oracle, since the specification can be evaluated to determine whether results are as specified or not. In the above running example, the following simple test (written in AsmL) will spot the error in *Reset*:

```
MyEnumerator e = new MyEnumerator();
while (e.MoveNext())
  Console.WriteLine("{0}", e.Current);

e.Reset(); // implementation error caught here
while (e.MoveNext())
  Console.WriteLine("{0}", e.Current);
```

6.6. CIL Weaving on CLR

The AsmL weaver is a tool that takes compiled AsmL and a compiled test subject and weaves them into a single CLR assembly. AsmL itself is implemented as a first-class CLR-hosted language. Thus, the weaver operates over two CLR assemblies, essentially merging condition checking code generated from the specification into the implementation code of the test subject. This approach enables the use of AsmL for specification and test of code implemented in any language hosted on the CLR.

7. Acknowledgements

I'd like to thank Mike Barnett, of the Microsoft Research FSE team, for providing the running AsmL example used in section 6.

8. References

Instead of providing a necessarily insufficient, incomplete, and in many ways unfair selection of references, I simply point to two recent books of mine that provide much deeper coverage and references.

[1] D.G. Messerschmitt and C. Szyperski, *Software Ecosystem – Understanding an Indispensable Industry and Technology*, MIT Press, 2003. (To appear.)

[2] C. Szyperski, *Component Software – Beyond Object-Oriented Programming*, second edition, Addison-Wesley, Harlow, England, 2002. (First edition, 1998.)