Final Project Report

Dynamic Assembly from Models (DYNAMO)

DARPA Dynamic Assembly for System Adaptability, Dependability, and Assurance Program

Agreement Number: F30602-00-2-0618

Spencer Rugaber
Georgia Institute of Technology
spencer@cc.gatech.edu

Kurt Stirewalt
Michigan State University
stire@cse.msu.edu

http://www.cc.gatech.edu/dynamo

October 30, 2003

Abstract

The DYNAMO project is concerned with assembling high-assurance systems from components, and, specifically, with guaranteeing correct interaction of sets of large, heterogeneous components. Several problems must be overcome to provide such guarantees: 1) dealing with the sheer complexity of the individual components and their interoperation; 2) maintaining design integrity and information hiding in the individual components; 3) providing the desired guarantees; and 4) not compromising efficiency while accomplishing the other goals. DYNAMO addresses these problems with several techniques: 1) a layered, implicit-invocation architecture limits complexity by reducing the quantity and nature of allowed interactions; 2) a declarative specification mechanism abstracts away low-level details such as event dispatch and handling and variable updates; and 3) compile-time component wrapper generation removes expensive, inter-layer procedure calls.

Acknowledgments

The PIs on this project wish to thank the following student participants: Corinne McNeely, Terry Shikano, Patrick Yaner, and David Zook from Georgia Tech and Reimer Behrends and AliReza Namvar from Michigan. We also wish to thank colleague Laura Dillon from Michigan State.

Executive Summary

The DYNAMO project is concerned with the high-assurance assembly of software components. A *component*[1] is a self-contained unit of computation capable of interacting with its environment by reacting to events, providing requested services, and managing its state. High-assurance is provided when a set of components, each of which acts according to its specification, results in a system that conforms to its specification. Specifications are expressed using the Object Constraint Language (OCL) a part of the industry-standard Unified Modeling Language (UML).

DYNAMO specifications describe invariant relationships among the states of an assembly's components. An *invariant* is a system property expressed in terms of the externally visible elements of its components' states. When an assembly receives an event from its environment, the state of one or more component can be altered. If that element of state is part of an invariant with other components, then it is necessary for those other components to be informed so that they might take steps to reestablish the invariant. This process is called *invariant maintenance*.

Invariant maintenance is difficult due to the sheer complexity of the components and their interactions. Moreover, invariant maintenance typically introduces costly run-time mechanisms for alerting components of changes. Even when invariant maintenance is achieved, confidence in system behavior remains low unless there is a convincing argument that invariants are properly maintained. Complex and voluminous code often makes such an argument difficult.

DYNAMO addresses the invariant maintenance problem with several techniques:

- **Model-based specification:** Invariants are specified in DYNAMO at a high level of abstraction by using the industry standard Object Constraint Language (OCL) a part of the familiar and well-supported Unified Modeling Language (UML).
- **Three-phased design:** DYNAMO supports high assurance via a design process that converts model-based specifications into a layered, implicit-invocation design guaranteed to maintained specified invariants. The design process consists of three phases that successively establish system context within its environment, decompose the system into components, and layer the components so as to provide high-assurance invariant maintenance.
- **Wrapper generation:** Confidence in expected system behavior is provided by the automatic generation of code supporting invariant maintenance. Generated code wraps existing components in a way that requires minimal intrusion into the components, thereby reducing the risk of introducing bugs.
- **Tool support:** The DYNAMO project has developed and adapted a variety of tools to support the specification, design, implementation and validation of high-assurance systems. Tools include a graphical UML modeling tool, an XML-based design extraction tool, an OCL parser, a code generator library, and a model checker.

The current status of the DYNAMO project is that the design process is well established and has been applied to a variety of small-scale problems including a text browser, a mail-spooler, and file version-control (briefcase) mechanism. The invariant maintenance architecture, including a number of variants is well-defined. And the tools exist in prototype form at various levels of maturity. Finally, a variety of interesting follow-up questions have been collected for future work.

1. Terms in italics are defined in the glossary at the end of this report.

Introduction and Problem Description

Software components are units of computation. They may exist in libraries or be specially constructed. Components can be assembled into systems. However, there is no guarantee that even correct components, when combined, result in correct systems. Invariants are desirable system properties. As a system executes and its state changes, the system must react to reestablish its invariants. This process is called invariant maintenance. A change of state may invalidate one or more invariants. For each such invariant, corrective actions must be located and verified. Unfortunately, this approach is error prone and expensive.

Invariant maintenance is difficult due to the sheer complexity of the components themselves and their interactions with each other. In addition to guaranteeing that system invariants hold, a solution to the invariant maintenance problem should satisfy the following additional properties. First, it must refrain, to the extent possible, from intruding into the components themselves. This property, sometimes called *transparency*, has several advantages. It separates reasoning about overall system properties from consideration of the individual components. Also, it reduces the need to modify the code of the components, thereby lessening the risk of introducing bugs.

The second property is *flexibility*. There are a variety of architectural mechanisms that support invariant maintenance. Flexibility enables the architectural approach to be selected by the designer based on other desirable system properties. Moreover, flexibility supports reuse, enabling components to be packaged in various ways.

The third property is low *overhead*. In particular, it should be the goal of any invariant maintenance mechanism to cause no additional run-time costs over an *ad hoc* implementation. As a general rule, the more encapsulated and self-contained components are, the more complex is the collaboration mechanism required to support them. With complexity comes run-time overhead. A low-overhead solution supports collaboration without any additional run-time cost.

The fourth property is *intentionality*. That is, in order to reason about system behavior, it should be possible to relate the behavioral specification of an invariant to its implementation in a direct way. In particular, each invariant should be traceable to the code mechanism responsible for maintaining it. Intentionality also supports maintainability—changes to system requirements often mean changes to the system's invariants. Intentionally implemented invariants are easier to alter.

The final property is *abstraction*. Confidence in system behavior is reduced when code must be examined to determine the effects of intricate combinations of events. Such efforts are labor intensive and error prone. DYNAMO uses the Object Constraint Language (OCL) [15] to specify system invariants at a high level of abstraction. Wrapper code is generated to ensure that invariants are appropriately updated.

The remainder of this section describes the assumptions DYNAMO makes about the problem space and work done by others that relates to component assembly problem.

Assumptions

DYNAMO is concerned with the high-assurance assembly of components. As a research project, we made several assumptions that limit the scope of the problems that we addressed.

- First of all, we are concerned with the assembly of interactive systems. Interactive systems are characterized by user-generated stimuli that produce visible responses. The complexity of such systems derives from the extensive state spaces that arise. We chose this area because of our experience with the MASTERMIND project in which we also used model-based code generation [22]. Interactive systems should be contrasted with data-intensive systems where the bulk of the complexity concerns the manipulation of fairly regular data. Although we did not experiment in this area, we believe that the DYNAMO approach could also be applied to reactive systems in which stimuli arise from sensors rather than users.
- DYNAMO is concerned with the assembly of systems with guaranteed execution properties. The primary type of property we investigated was behavioral properties. Such properties concern correct execution; that is, execution that produces expected output. We did some investigation of synchronization properties as well. Synchronization properties indicate how components collaborate to produce results. We did not explore quality-of-service properties.
- The applications that we looked at were primarily single threaded. That is, control was sequential within a single address space. We did one experiment with single process, multi-threaded applications. We did not look at multi-process applications.
- The design scenario we addressed is system assembly from components. The components may exist in a library or they may be built from scratch. We are not concerned with monolithic systems.
- Finally, we have concentrated on systems written in the C++ programming language. C++ features a powerful generic facility (called templates) that supports *metaprogramming* and efficient compilation, two features we wanted to exploit. We believe that DYNAMO's conceptual contributions could be applied to any language with these features.

Related Work

Context. There are a variety of design strategies for maintaining invariants among an assembly of components. At one extreme, an invariant can be implemented as an explicit integration component, distinct from the components it integrates (hereafter referred to as *integrands*). Under this approach, the integration component might be a peer of its integrands, as is the case with mediators[23], or it might *encapsulate* its integrands, as with GenVoca layers [2] and facades and adapters [8]. Some designs even employ a hybrid of these approaches. For example, Java AWT programmers define containers, which (like layers) encapsulate GUI components but which (like mediators) listen for events from these components [11]. At the other extreme, an invariant can be implemented as a *collaboration* [24][17], which distribute the responsibilities for maintaining the invariants among the integrands. The choice of integration strategy is subject to numerous trade-offs. If we choose to use an integration component, encapsulation can usually be implemented more efficiently. Integration components have the advantage of not requiring integrands to be modified with invariant-specific code; however, collaborations, by virtue of being within the integrands, can use knowledge of an integrand's internal state to implement a more efficient update protocol. These trade-offs can only be assessed once the invariants are known, i.e., at assembly time.

In practice, it is difficult to exploit this trade-off at assembly time because integration strategies largely depend upon the mechanisms that integrand components use to communicate with other components in the assembly. The collaboration approach uses *explicit invocation*, which (in an OO implementation) requires an integrand to store a reference to the other integrands and to directly invoke operations through these references. By contrast, the mediator approach relies on *implicit invocation*, which requires the changed component to announce an event whenever its state changes in a way that might trigger the reestablishment of an (as yet unknown) invariant. Finally, under encapsulation, an integrand should not initiate communication with other components at all! The key problem then is how to design components so that the choice of mechanism for communicating with other components can be delayed until assembly time. This problem, which is also called the *flexible packaging problem* [6] requires a two-fold solution. First, a component must be designed to communicate with its environment using a mechanism that is more abstract than the traditional procedure call. Second, a component must be *packaged* prior to insertion into an assembly. Packaging involves replacing the abstract communication primitives in a component's source code with actual code that implements these primitives.

Beugnard et al. Beugnard et al. [4] are concerned with high assurance component interactions. Component interactions are described by contracts clearly specifying expected interactions. Of particular interest to the DYNAMO project was the authors' breakdown of contracts into four categories.

- **Basic (or syntactic) contracts** provide names for operations, parameters, and exceptions. Basic contracts are what are specified by interface description languages, such as IDL [12], a part of CORBA.
- **Behavioral contracts** assign responsibilities to components. Behavioral contracts are what are normally specified by pre and post conditions expressed in predicate logic.
- **Synchronization contracts** describe allowable coordination policies between components. Path expressions [5] are a common notation for expressing synchronization contracts.
- **Quality of service contracts** describe how non-functional requirements, such as resource consumption and performance are handled by a set of components.

Most of the DYNAMO work has been concerned with behavioral contracts. We have used the Object Constraint Language [15][25] (an extension to UML) to express these contracts. We have also done some work investigating synchronization contracts. For this we have developed a powerful model of synchronization contracts called the *Universe Model* [3]. We have also used CCS [14], Milner's notation for specifying intercomponent synchronization behaviors to formalize our approaches to invariant maintenance.

Shaw and Garlan. Shaw and Garlan, in their book on software architecture [21] describe a design space for implementations of implicit invocation mechanisms. *Implicit invocation* provides one solution to the invariant maintenance problem in which components whose status changes notify dependent components which have expressed their interest. The notifying component is not explicitly aware of the identity of the notified component. This design space is also discussed in the article [9]. Among the factors the Shaw and Garlan consider are the following.

1. **Fixed or dynamic vocabulary for events:** Whether the set of notification events is fixed in advance. For DYNAMO, the set of events is determined by the set of status variables, which is determined for the assembly specification at compile time.
2. **Built in or user defined:** Whether the set of events is fixed by the implementation or user defined. For DYNAMO, the set of events is fixed by the set of status variable being monitored. Changes to the status variables trigger a notification event to dependent components.
3. **Explicitly declared or not:** Whether the set of events must be explicitly declared or is dynamically determined. For DYNAMO, the set of events is inferred, at compile time, from the set of status variables being monitored.
4. **Central or distributed declaration:** Whether the event declarations are collected in one place or distributed throughout the assembly. From the point of view of the designer, events are associated with status variables, and are therefore distributed throughout the assembly.
5. **Parameters:** Events are atomic notifications, but it is often useful to supply values with events as parameters. For DYNAMO, event parameters may be requested from notifying components via service requests (method calls).
6. **Registration required:** Whether or not *a priori* event registration is required. For DYNAMO, the set of events to be handled is collected from the specification at compile time, as are the sending (independent) and receiving (dependent) components. Registration is effected by the dependent component providing a notification method to be invoked by the independent component when a status change occurs.
7. **Translation of event parameter to method parameter:** Most programming languages do not have an event primitive. So event notifications must be translated into method calls. Moreover, event parameters must be made available in the method call. For DYNAMO, this is done at compile time.
8. **Announcement mechanism:** How are event notifications implemented? For DYNAMO, each notifying component aggregates a set of status variables, the values of which other component depend on. For each status variable, a list of notification methods in maintained. Changes in the values of status variables, cause these methods to be invoked, thereby notifying dependent components.
9. **Component implementation:** What programming language element is used to implement a component? For DYNAMO, a class is used to implement a component. However, a collection of components that reside in the same layer in the architecture can be implemented as nested classes within a master class for that layer.
10. **Concurrency:** Whether components and events are treated concurrently or in a single thread. Most of the DYNAMO work has been concerned with a single thread of execution. However, we have conducted some experiments in which multiple threads in a single process were used.
11. **Delivery policy:** Under what circumstances and in what order are dependents notified? For DYNAMO, all dependents are notified; the order is fixed at compile time. Shaw and Garlan call this *full delivery.*

Dingel, Garlan, Jha, and Notkin. Components in DYNAMO assemblies inform each other of state changes using implicit invocation. Implicit invocation is an architectural style in which state changes in one component are announced to other, dependent components, implicitly. That is, the announcing component is not aware of who it is notifying. Dingel *et al*. have provided a formal-

ization of this process using process algebra and trace semantics [7]. Their approach is compositional—it supports reasoning about composed systems by combing the reasoning about the components. Although DYNAMO is primarily concerned with implementation approaches for implicit invocation systems, we have experimented with formalization, using Milner's Calculus for Communicating Systems [14].

Riehle. DYNAMO uses C++ templates to implement implicit invocation. Riehle has also designed a C++-based template solution to this problem [18]. His approach is called the *Event Notification Pattern*, related to the Observer design pattern in the design patterns book [8]. In particular, Riehle makes the same, asymmetric design choice as DYNAMO, supporting implicit notifications from lower-layer components to upper-layer components, but requiring notifications in the other direction to be explicit. One major difference between Riehle's approach and that of DYNAMO is that with the former, components must explicitly indicate relevant state changes, whereas, with DYNAMO, state changes are automatically detected and communicated.

VanHilst and Notkin. A *collaboration* is a protocol of interaction among multiple objects (called *roles*) to achieve some purpose or goal or to implement some invariant [24]. Here, a role is not an actual object, but rather a fragment that comprises the subset of an actual object's characteristics that are required for the object to participate in a particular collaboration [17]. Moreover, a role might be abstract, which is to say that it merely declares some operations without providing methods to implement them. In fact, the most reusable collaborations are those for which one or more of their roles is abstract. We can think of a collaboration as a collection of roles that interact according to a protocol of message exchange, and we can visualize this behavior using a UML sequence diagram, each of whose columns is labeled by a role.

Because roles correspond to fragments rather than actual objects, role types are difficult to express as first-class entities in many programming languages. VanHilst and Notkin [24] describe how to represent roles in C++ using *mixin classes*, which are class templates in which the template parameter is used to define the base class from which the (instantiated) mixin class derives. By defining roles in this manner, a designer can extend a class to play a new role by instantiating the mixin class that implements the role with the class to be extended.

For example, suppose we have a class `C` that we want to adapt to play role (R_1) in a new collaboration. If $\texttt{role}R_1$ is the mixin class that implements R_1, then we can adapt `C` to play this role by defining a new class:

```
C'= roleR1<C>
```

In addition, if we want `C` to play roles R_2 and R_3, we would declare `C'` as follows:

```
C' = roleR3< roleR2< roleR1< C > > >
```

In fact, using this technique, one can essentially derive a custom class just by identifying the roles its objects will need to play in various collaborations.

DYNAMO is concerned with the assembly of systems from interactive components and declarative specifications of assembly guarantees, which govern component interaction. Ultimately,

these declarative specifications must be translated to code that interfaces with the components being integrated. Because an assembly guarantee is an invariant, it can be thought of as a collaboration, each role of which is played by one or more of the components being assembled. We have thus explored how to implement assembly by generating mixin classes for each role in the collaboration denoted by an invariant and then extending each component to play the appropriate role using mixin-class instantiation.

Batory and Smaragdakis. Mixin classes allow the explicit definition of roles in languages that support parameterized inheritance. Smaragdakis and Batory [20] extended this idea to represent collaborations using a structure called a *mixin layer*, which defines a collaboration to be a template class with one or more nested classes, each of which is a mixin class that defines a different role in the collaboration. Unlike mixin classes, the nested classes in a mixin layer do not assign a new name to the role being defined, but instead refer to the class that is being refined to play the new role. A mixin layer has the following form (in C++):

```
template <typename PARENT> {
class CollaborationC1 : public PARENT
    class A : public typename PARENT::A {
    ...
    };
    class B : public typename PARENT::B {
    ...
    };
    ...
}
```

Notice that each nested class (e.g., `CollaborationC1::A`) refines a given class (e.g., `A`) to play a role in C_1. Using mixin layers, one can synthesize a collection of classes by composing the collaborations in which the objects of those classes will play a role.

Mixin layers were originally developed to implement GenVoca layers and layered composition without requiring the development of a program generator. In the original GenVoca paper [2], a component is defined to be a highly-cohesive collection of classes. Layered composition then causes the simultaneous refinement of multiple classes. In DYNAMO, we use mixin layers and layered composition as a means to represent a specially designed component called a mode component, which is like GenVoca layers that also announce events. Most assembly invariants can be implemented directly using one or more mixin layers, which are then used to simultaneously refine a collection of components to play roles in the assembly.

Sullivan and Notkin. A *mediator* is a software component that reifies integration relationships into a component that is separate from the components being integrated [23]. Communication between a mediator and its integrands is asymmetric in that integrands communicate with the mediator indirectly, using implicit invocation, whereas the mediator communicates directly with integrands using explicit invocation. This asymmetry is motivated by the desire to maintain component independence in order to simplify evolution. Consequently, each integrand must announce

an event whenever any potentially important change in state occurs. A mediator integrates a collection of integrands by registering for events in each integrand and then, upon receiving events, alerting dependent integrands to update their state accordingly. Integration via mediators is the most general and most flexible strategy for component assembly; however, this generality and flexibility is achieved via a heavy use of indirection, which may incur unacceptable performance penalties when the technique is used to integrate more fine grained components. In DYNAMO, we use mediators to implement a system of assembly invariants that incurs a dependency cycle, and we use them in conjunction with encapsulation to construct mode components.

De Line. A major inhibitor to the reusability of software components concerns the communication mechanisms a component uses to interact with its dependents. Suppose, for example, that component C_1 uses a service of another component C_2. If these components are to reside in the same address space, then C_1 will communicate with C_2 using a method call; whereas if they reside in different address spaces, C_1 must use a remote procedure call. Unfortunately, in most programming languages, the choice of communication mechanism is bound at component development time, when in fact we would often prefer to defer the decision to assembly time. *Flexible packaging* is concerned with how to design software so that the choice of communication mechanism can be delayed until assembly time. DeLine proposes a solution in which components communicate indirectly using channels via Linda-like [10] coordination primitives [6]. In the Ciao integration language, channel communication can be replaced with procedure calls, remote-procedure calls, and relational-database queries, thus allowing the packaging decisions to be deferred to assembly time. DYNAMO uses a more restricted form of flexible packaging, based on C++ template instantiation, which allows the choice of integration strategy to be deferred to assembly time.

Results

Model-based specification

DYNAMO supports model-based specification of component assemblies. What this means is that a designer specifies an assembly in a high-level, declarative notation rather than operationally in a programming language. The notation we have used is UML (the Unified Modeling Language) including the Object Constraint Language (OCL). In particular, we have interpreted UML class model constructs in terms of the vocabulary of software architecture. The interpretation is presented in Table 1. Annotations to the class model, in the form of OCL constraints, provide semantics. In particular, external system events (stimuli) are ultimately modeled as methods in a component. OCL pre and post condition constraints specify the effect of an event on the system. Invariants, initially indicated with natural language annotations, are first translated by the designer into OCL associations. As the architecture is refined, associations are subsumed by DYNAMO's layered architecture. At this point, the constraints are assigned to the component responsible for maintaining them.

Table 1: DYNAMO UML Interpretation

UML Concept	DYNAMO Interpretation
System	Assembly
Package	Layer
Class	Component
Attribute	Percept
Association	Invariant
Dependency	Event

There are several benefits that derive from using a declarative notation. Because OCL is high-level and declarative, designers can concentrate on system properties rather than worrying about details. Because it is abstract, it is more concise than code, reducing the maintenance burden. Because it is declarative, error-prone procedural details are elided. Finally, because OCL is formally defined, it supports reasoning. That is, OCL constraints can be used to check system properties using various existing tools.

Three-phased design method

The DYNAMO design method[1] starts with a declarative model of an assembly expressed as an annotated UML class diagram using a graphical CASE tool. The diagram is refined, first into loosely coupled components that are then organized into a layered architecture. From the resulting UML model, C++ wrapper classes can be generated that assemble the components. To support efficiency and reuse, components are assembled using a layered, implicit-invocation architecture called a *mode-component architecture*. A *mode component* is a specialized component that alerts its clients when its state changes. Additionally, the correctness of these generated assemblies can be verified either statically, using tools such as theorem provers or model checkers, or dynamically, by run-time assertion checking.

1. Details of the DYNAMO design method can be found in reference [13].

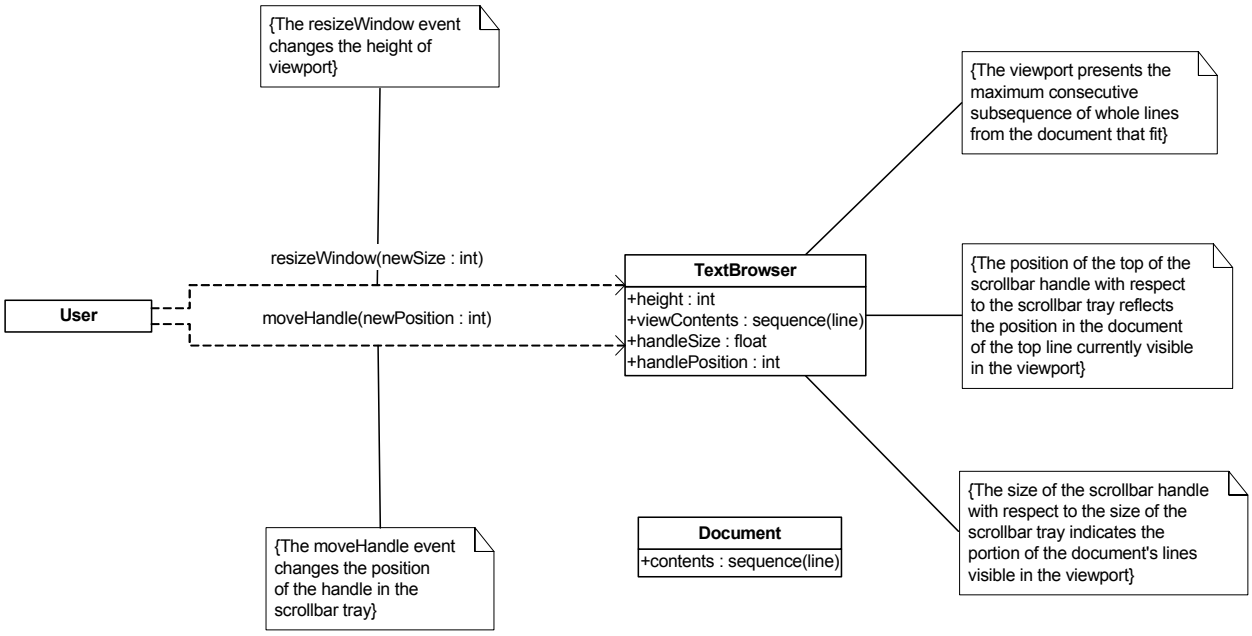

Figure 1: Phase 0 Diagram for Text Browser Example

The DYNAMO design method comprises three phases that refine a conceptual model of a proposed assembly into interrelated components organized as layered mode components. In Phase 0, the environment in which the assembly executes is described in terms of external actors, the assembly itself, the communication among them, and the behavioral properties (invariants) that the assembly must guarantee. Phase 1 asks the designer to partition the assembly into its constituent components and their relationships, assigning responsibility for external actions and invariant-maintenance to the components appropriately. Finally, Phase 2 asks the designer to layer the constituents as mode components, where lower-level components communicate status changes upward, and higher-level components make specific service requests of lower-level components. Phase 0, 1 and 2 diagrams for a simple text browser assembly are presented in, respectively, Figure 1, Figure 2, and Figure 3.

In Figure 1, the assembly is denoted by the `TextBrowser` icon. Two other actors comprise its environment—the user and the document to be viewed. User interactions are denoted with dependency arrows, and guarantees are provided within annotation icons.

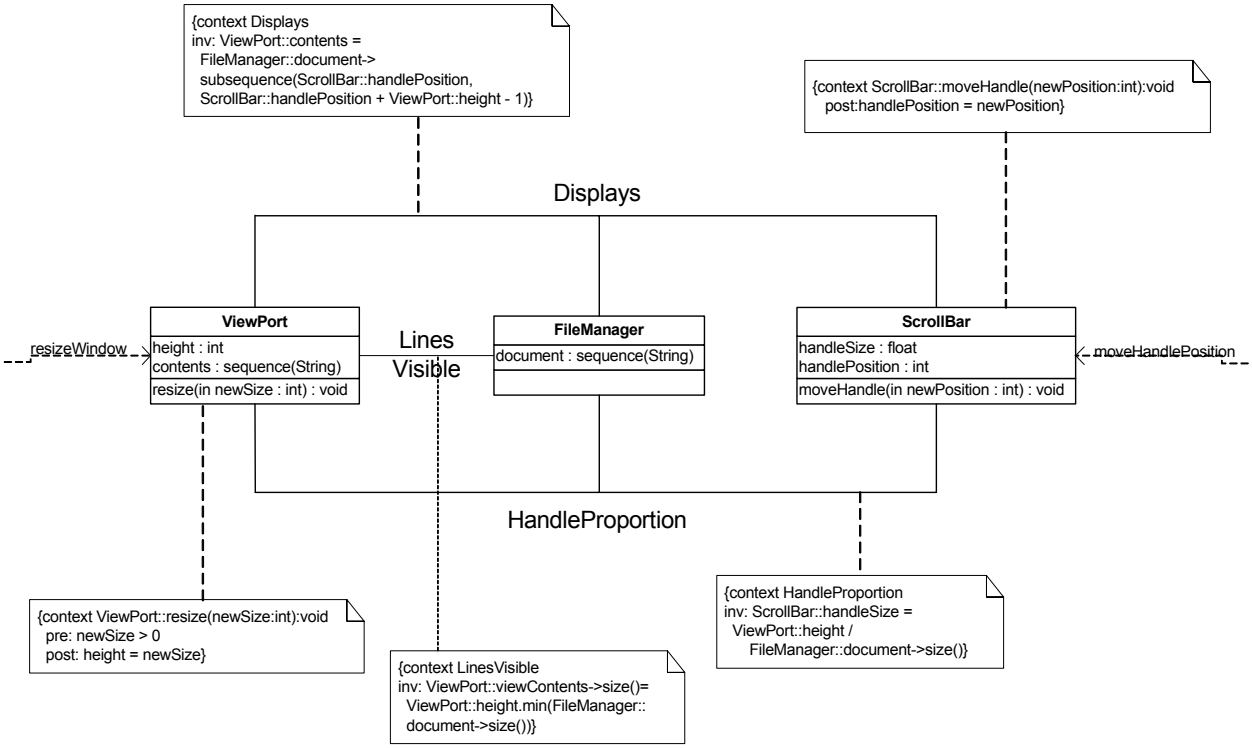

Figure 2: Phase 1 Diagram for the Text Browser Example

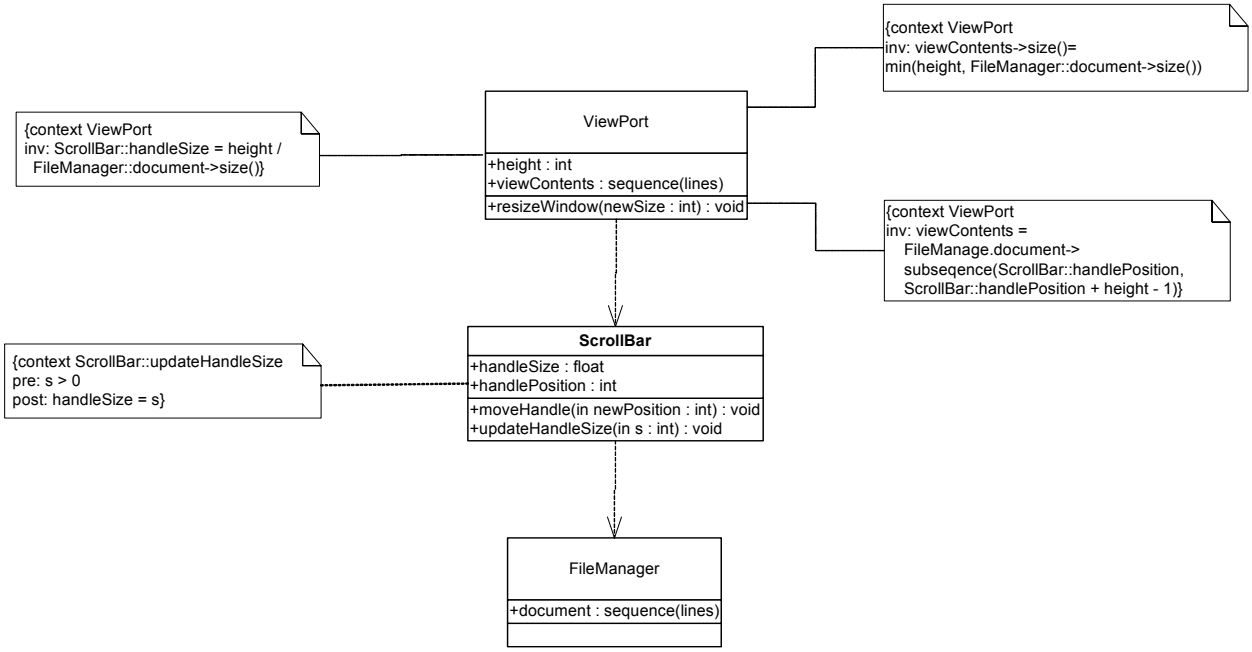

Figure 3: Phase 2 Diagram for the Text Browser Example

In Figure 2, the TextBrowser has been decomposed into three components: a viewport, a

scrollbar and a file manager. Three associations have been included to indicate the collaborations required to enforce the `TextBrowser`'s guarantees. The associations have been annotated with OCL constraints denoting the invariants the assembly must maintain. The two user events have been delegated to components as have the four assembly percepts.

Figure 3 presents the layered, implicit-invocation architecture used to effect invariant maintenance. The three components have been organized into a stack, and the OCL constraints have been assigned to the respective dependent attributes. Ultimately, the components will be realized as wrapped and nested template classes.

Tool support

A variety of prototype tools were built or adapted to support the investigation into invariant maintenance. The DYNAMO tools architecture is shown in Figure 4. The designer using DYNAMO

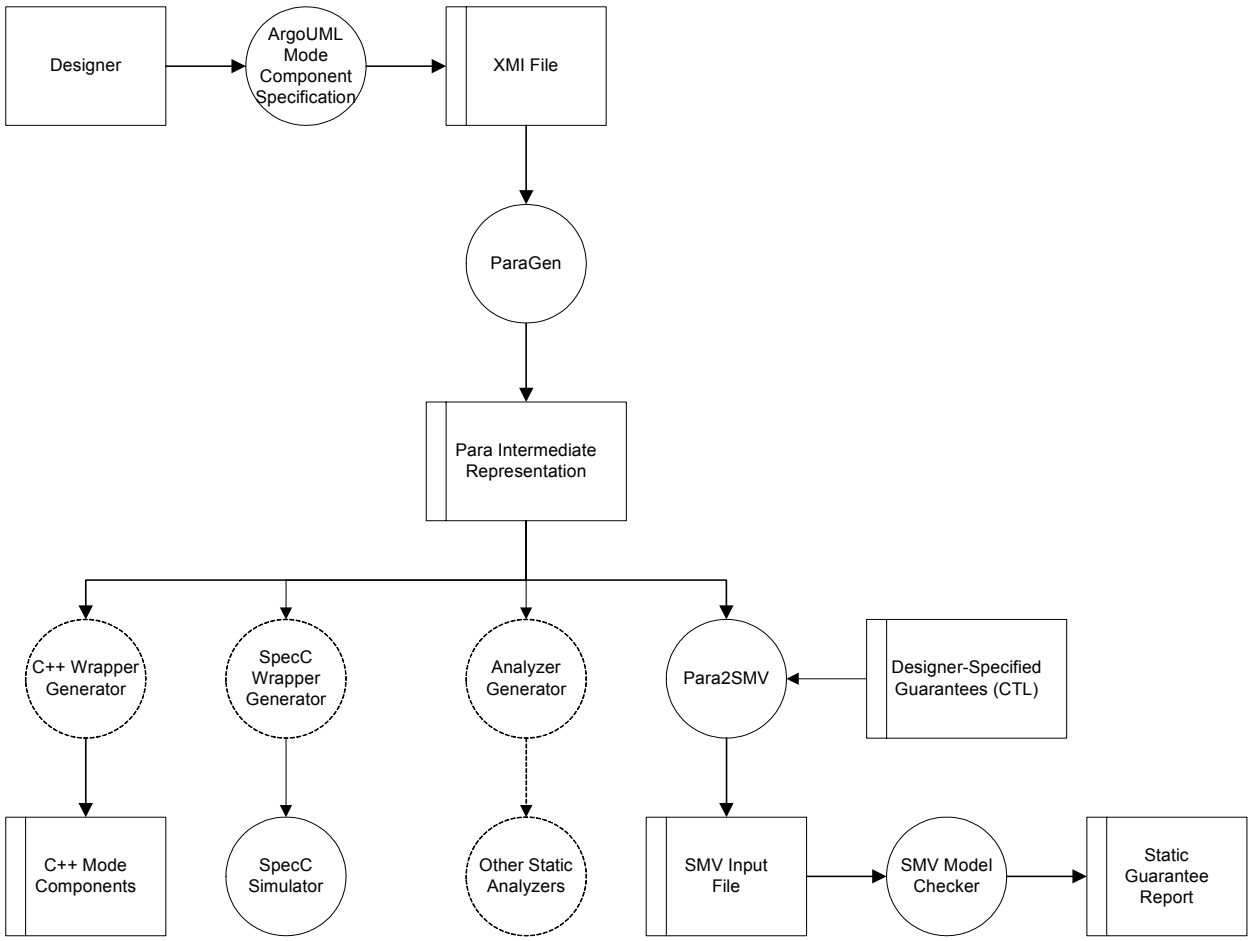

Figure 4: DYNAMO Tools Architecture

tools might begin with a UML class model drawing tool such as ArgoUML to construct a DYNAMO design. ArgoUML is able to export such designs using XMI, an industry standard XML schema for UML CASE tool design descriptions. We have built a tool called ParaGen for extract-

ing design information from an XMI file. Two version were constructed. The first was a stand-alone C++ program. It made use of the Xerces XML parser from Apache. The second used XSLT. XSLT is a language for manipulating XML data. XSLT programs describe transforms that can be applied to XML to construct output files in either XML, HTML or text. We used the Xalan tool from IBM to process the XSLT transformations. For both the C++ program and the XSLT program, we produced output in a format called Para, which is the DYNAMO intermediate representation.

The intent of the Para intermediate representation is to make it easy to apply a variety of tools to a DYNAMO design. We tried two specific applications. First, in order to do static analysis of UML designs, we applied the SMV model checker (from Carnegie Mellon and Berkeley) to them. In this case, the designer augments the class model diagram with a statechart and generated XMI from the diagrams. Then, Paragen is used to convert the design into Para format.

We built a tool called Para2SMV for converting designs in Para format into the input language for SMV. The designer must add a CTL (Computational Tree Logic) description of the property to be checked. Then SMV can be run to see whether the property holds for the design. To support this generation process, we built a C++ library, called `SmvModel`. These classes are used within Para2SMV to construct the SMV input file, but they are written in such a way that they can be used in any application that needs to construct SMV.

The other tool that we built was the COGITO mode-component compiler, illustrated in the lower left hand corner of Figure 4. If the designer has included OCL constraints in the UML class model diagram, these will be included in the generated XMI file and conveyed to the Para representation. COGITO parses the OCL and produces a parse tree internal representation. We have constructed a C++ library, similar to `SmvModel`, called `C++Visitor` that can generate C++ mode component wrappers for components described in the UML class model diagram. The wrappers, when linked with the original components maintain specified system invariants, hence providing assured behavior.

The other components in Figure 4 have not been built. They are shown in the diagram to illustrate how the tools architecture might be extended to support other languages and other static checkers.

Efficient implementation of guaranteed behavior

A *wrapper* is a code fragment that can be used to adapt an existing component for a variety of purposes such as providing a different interface or enhanced functionality. DYNAMO mode-component wrappers detect alterations to component status and inform dependent components. If the mode component is itself dependent on other components, then the wrapper also provides the code that reestablishes invariants to reflect the changes to status.

Mode component wrappers are implemented using several features of the C++ programming language[1]. One feature of interest is operator overloading. That is, programmers can provide new interpretations for built-in C++ operators. In the case of DYNAMO, the assignment operator is overloaded. When an assignment is made to an element of component status, a DYNAMO-gener-

1. Details of mode component implementation can be found in reference [19].

ated method is invoked that, besides performing the assignment, notifies dependent components. This additional ability is provided without altering the source code that performed the assignment.

Alternative invariant maintenance mechanisms

Besides mode components, we have explored a variety of alternative mechanisms for implementing invariant maintenance. There are two goals for this exploration: 1) to apply DYNAMO code generation techniques to these mechanism and judge the extent to which their implementations can benefit from them; and 2) to better understand the design space of invariant maintenance mechanisms and, ultimately, to produce a policy-based implementation [1]. In the remainder of this subsection, we present three alternative invariant maintenance mechanisms with which we have experimented.

Encapsulated collaborations. To compare the different invariant maintenance strategies, consider a running example in which three components, `a`, `b`, and `c` collaborate to maintain the invariant `a = b + c`.

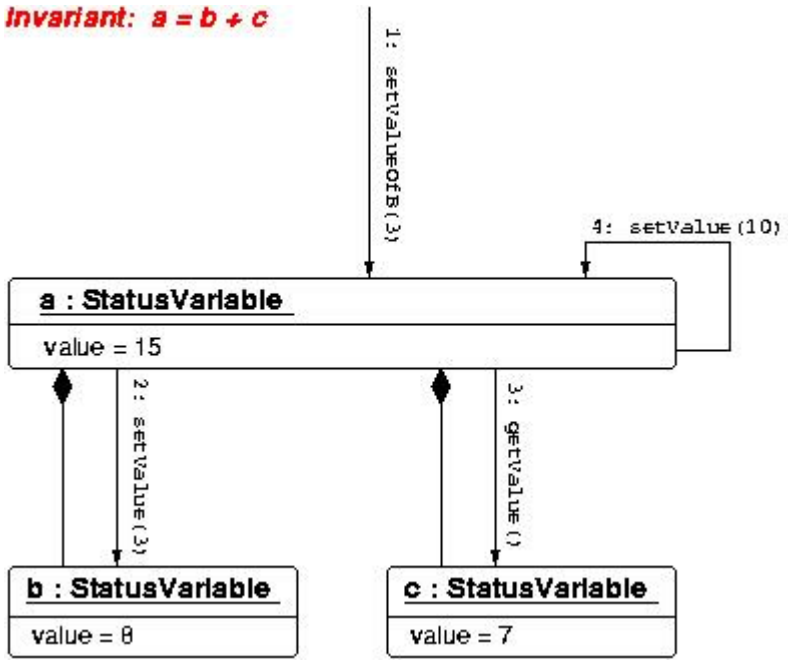

Figure 5: Encapsulated Invariant Maintenance

Figure 5 depicts a collaboration that maintains this invariant when the components are assembled using encapsulation. Notice, because `a` encapsulates `b` and `c`, all external accesses to `b` and `c` must go through `a`. Moreover, `a` is responsible for maintaining the invariant. Specifically, when `a` receives the message `setValueOfB(3)`, it sends the message `setValue(3)` to update the value of `b`, then retrieves the current value of `c`, and sets its own value to the sum. This approach does not require any modification to the aggregated components (i.e., `b` and `c`), but it requires significant modification to the aggregator (`a`). In the worst case, the interface of the aggregator could swell to include the union of services over the interfaces of all of the aggregated components.

Moreover, every invocation of a service that might modify one of the aggregated components incurs the cost of an invocation of all of the other aggregated component in order to retrieve values necessary to re-establish the invariant.

Mediated collaboration. \Figure 6 depicts a collaboration that maintains the same invariant using a mediator.

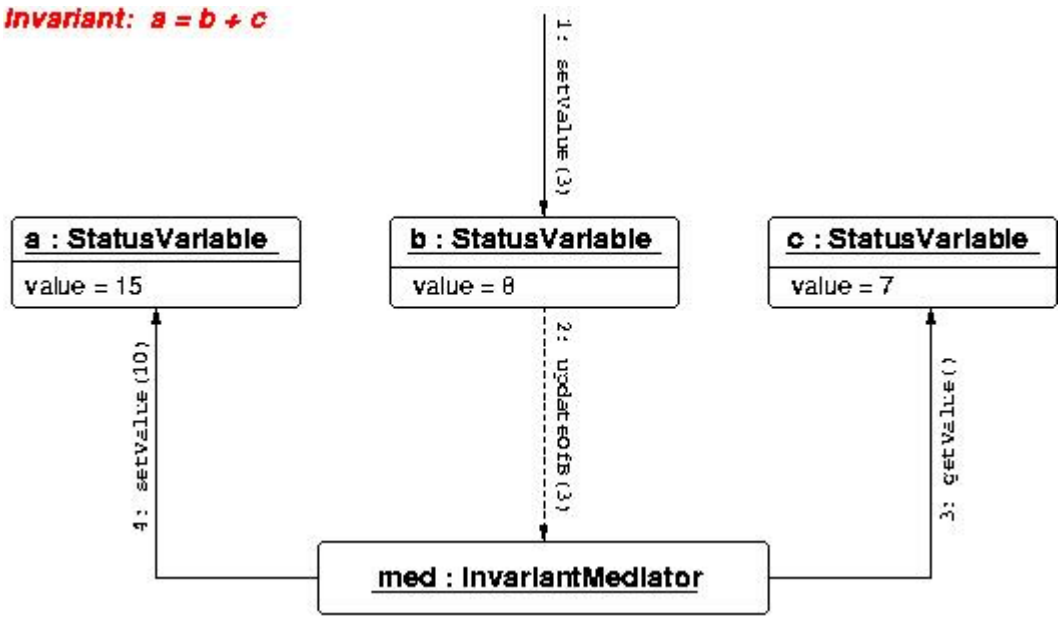

Figure 6: Mediated Invariant Maintenance

In contrast to Figure 5, collaborating components are peers, and each is visible to other clients who might use its services. In Figure 6, such a client sends the message `setValue(3)` to component `b`. In response, `b` sends the message `updateOfB(3)` to the mediator, which is responsible for maintaining the invariant. The mediator handles this message by first retrieving the value of component `c` and then setting the value of component `a` to be the sum. As with encapsulation, one component (the mediator) knows about all of the other components, but the mediator is a new component that does nothing but maintain the invariant. Consequently, the mediator must provide an `update` operation for each component; whereas with encapsulation, these operations are part of component `a` (as is the invariant-maintenance logic). Also like encapsulation, any update of any component incurs messages to all of the other components in order to retrieve values needed to reestablish the invariant. Unlike encapsulation, each of the components must know about the mediator.

Distributed collaboration. \Figure 7 depicts a collaboration that maintains the same invariant when the components are assembled using a many-to-many distributed collaboration.

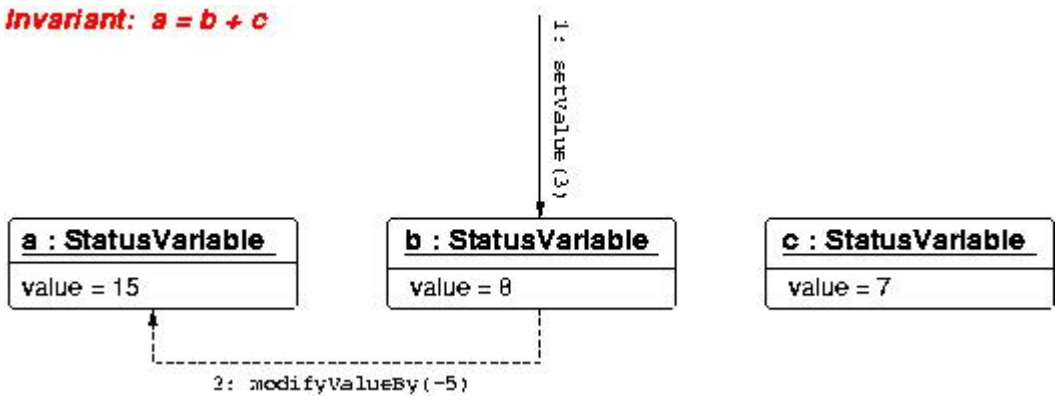

Figure 7: Distributed Invariant Maintenance

In this case, components `b` and `c` contain direct references to component `a`. Notice, however, that the responsibility for invariant maintenance is now distributed rather than localized. For example, when `b` receives the message `setValue(3)`, rather than announcing its new value to `a`, it announces the difference between the new and old values, and `a`'s implementation of the `modifyValueBy(-5)` message adds this delta to its current value. Such a distribution has the advantage of optimizing the number of messages, but it does so at the expense of distributing knowledge of the invariant among the collaborators.

Implementation of alternative invariant maintenance mechanisms

The three different composition strategies are supported by a hierarchy of classes, some of which are assembly dependent. Our implementation library provides the following six reusable classes:

- **`StatusVariable`** is a parameterized and abstract class that generalizes all status variable objects in a system. The class provides a polymorphic `getValue` operation but no facility for setting the value, as some values will be computed on demand or set implicitly.
- **`StatusVariablePrimitive`** is a parameterized concrete class that extends `StatusVariable` with an explicit `setValue` operation.
- **`StatusVariableUpdate`** extends `StatusVariablePrimitive` with facilities for registering and announcing updates to one or more listeners. Objects of this class respond to `setValue(v)` messages by updating their value and then sending the message `update(v, this)` to all registered listeners, where `this` is the object that received the `setValue` message.
- **`StatusVariableDelta`** extends `StatusVariablePrimitive` with facilities for registering and announcing changes (i.e., the difference between the old and new value) to one or more listeners. Objects of this class respond to `setValue(v)` messages by (1) computing the difference `delta = v - val` where `val` is the old value, (2) setting `val` to `v`, and (3)

sending the message `modifyBy(delta, this)` to all registered listeners, where `this` is the object being updated.

- **`StatusVariableListener`** is an interface class that declares the `update` message. Objects that implement this interface can register as listeners to `StatusVariableUpdate` objects.
- **`StatusVariableDeltaListener`** is an interface class that declares the `modifyBy` message. Objects that implement this interface can register as listeners to `StatusVariableDelta` objects.

Encapsulated Invariant Maintenance. Figure 8 depicts the classes used to implement encapsulated collaboration. Notice there is only one assembly-specific class, `StatusVariableA`, which aggregates two primitive status variables and provides setter operations `setValueOfB` and `setValueOfC`. Instances of `StatusVariableA` handle the messages of the form `setValueOfB(v)` by invoking `b.setValue(v)` and then updating the local value in accordance with the invariant. `setValueOfC` messages are handled similarly. By virtue of encapsulation, clients can only access instances of `StatusVariableA`.

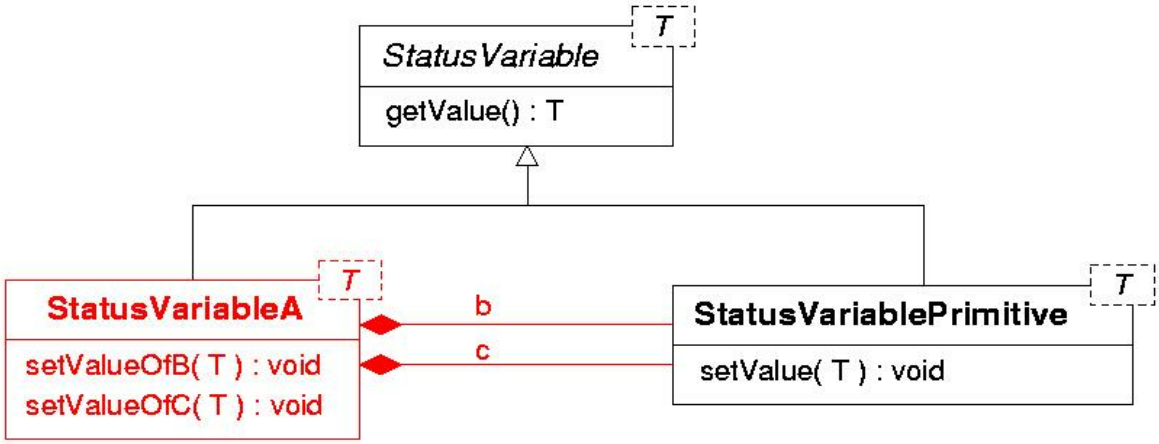

Figure 8: Implementation of Encapsulated Invariant Maintenance

Mediated Invariant Maintenance. Figure 9 depicts the classes used to implement mediated collaboration. In this configuration, status variable `a` is an instance of class `StatusVariablePrimitive`. By contrast, `b` and `c` are instances of class `StatusVariableUpdate`, which means they announce all updates to any registered listeners. The invariant is implemented by an instance of the (assembly-specific) class `Mediator`. By implementing the `StatusVariableListener`, instances of class `Mediator` can register for and receive updates from the `b` and `c` objects. Notice also that class `Mediator` contains references to each of `a`, `b`, and `c`, which allows it to get and set values as appropriate to maintain the invariant. Moreover, upon receiving a

message of the form `update(v, o)`, a mediator can compare `o` with `b` and `c` to decide which object sent the message.

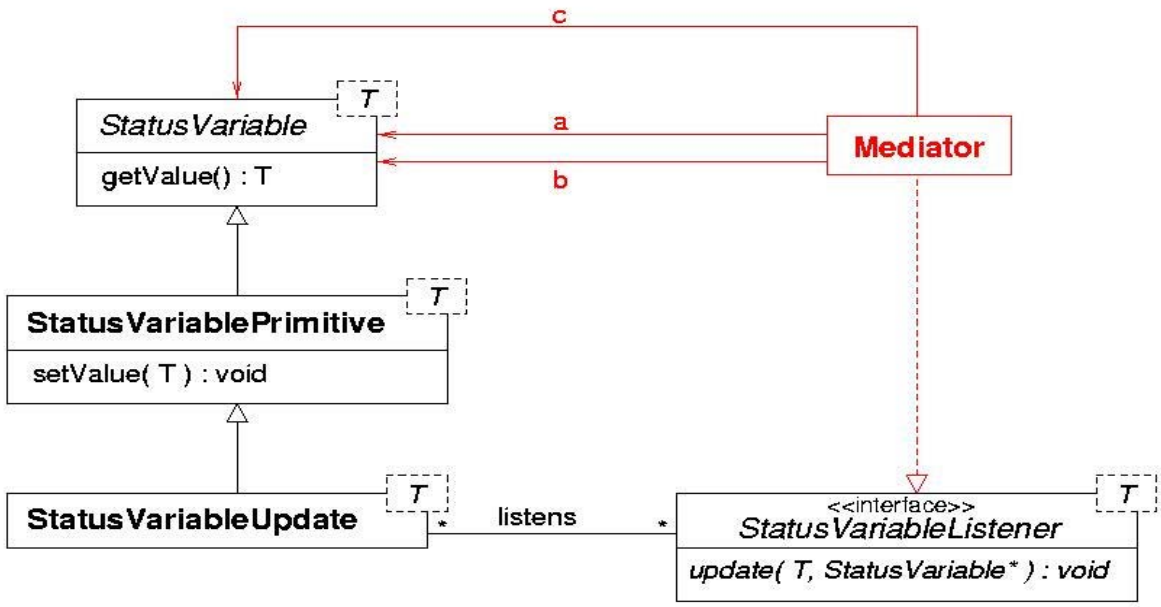

Figure 9: Implementation of Mediated Invariant Maintenance

Distributed Invariant Maintenance. Figure 10 depicts the classes used to implement distributed collaboration. In this configuration, the status variable `a` is an instance of an assembly-specific class `StatusVariableA`, which references two status variables (`b` and `c`) and implements the `StatusVariableDeltaListener` interface in order to receive `modifyBy` messages from

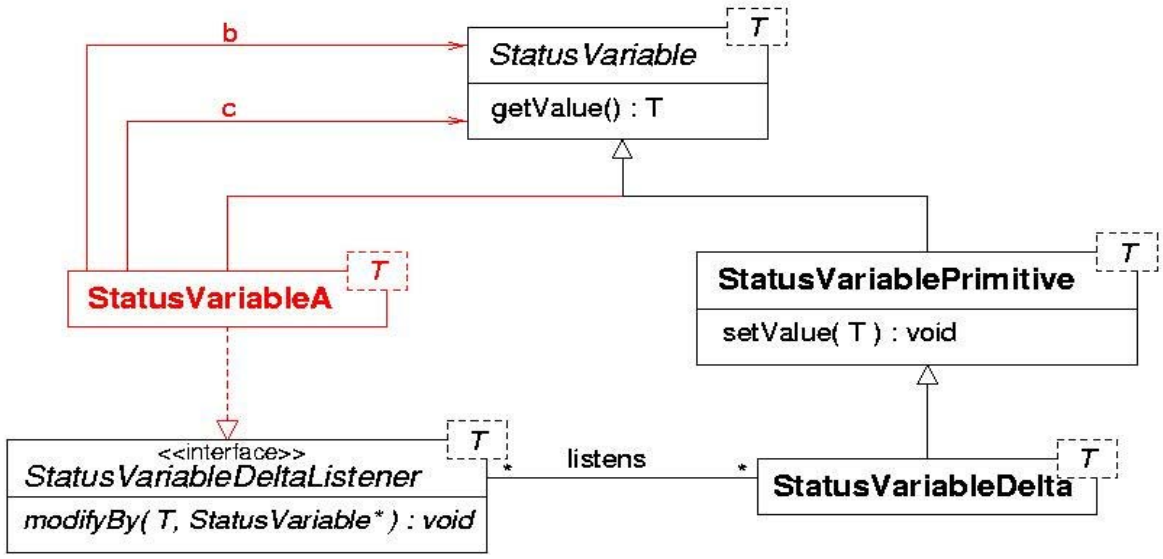

Figure 10: Implementation of Distributed Invariant Maintenance

the objects that implement `b` and `c`. Variables `b` and `c` are instances of the class `StatusVariableDelta`.

Future Work

CCS formalization of alternative invariant maintenance mechanisms

A software component defines services in its interface and implements these services by requesting services from other components. In addition to providing and using services, an interactive component announces events to communicate changes in state to interested observers. Service requests and event notifications are ultimately realized by inter-object collaborations, but there are many different ways to implement these collaborations. In a large assembly, the designer should be allowed to choose among different strategies in a disciplined manner. We designed DYNAMO components to be adaptable in this fashion, so that one can make these packaging and coordination decisions at assembly time, when one is better able to balance the costs and benefits of these decisions.

The flexibility afforded by this design allows us to compose components using a variety of higher-level' assembly operators—such as *encapsulation*, *peer-to-peer (distributed) communication*, and *mediated communication*—and, in fact, we anticipate that designers will be able to specify assemblies by selecting a group of components, selecting an assembly strategy, and then pushing a button to generate the code that implements the assembly. To make these ideas precise, we formalize DYNAMO components and these higher-level operators by modeling them in the CCS notation [14]. In this formalization, DYNAMO components are CCS agents, and the assembly operators are idiomatic combinations of CCS operations applied to these agents. Moreover, we define a transparent implementation of these CCS composition idioms, thereby allowing an

assembly designer to manipulate components in a formal calculus and have correct code generated from these specifications.

Finite differencing optimizations

DYNAMO uses invariants to generate assembly-specific code from which to compose multiple, collaborating components. Because these invariants are not known until assembly time (i.e., after the design and implementation of the individual components), this invariant-maintenance responsibility must be discharged by introducing new code somewhere in the assembly implementation. Often many of the collaborating components must be extended, and the nature of the extension may depend upon the particular invariant. For example, suppose a distributed collaboration maintains an attribute (e.g., `a`) in one component as the sum of attributes (e.g., `b` and `c`) in other components. Then any change to the value of `b` or `c` requires notifying `a` of the change, and `a` then recomputes the invariant expression. Because recomputing the invariant expression may incur multiple messages among these distributed components, we may instead wish for `b` to include, in the notification, the difference between the old and new values for `b`. This way, `a` can update its value without reevaluating the invariant expression.

Finite differencing is a program optimization that systematically replaces expensive computations of an applicative expression $E = f(x_1, x_2, \ldots, x_n)$ with an explicit variable that maintains the value of `f` and additional code that updates the variable (without recomputing `f`) when one of the dependent variables (i.e., $x_1, x_2, \ldots, x_n$) changes [16]. We borrow ideas from this approach to generate invariant-maintenance code in the components that contain spoiling attribute updates (i.e., components that update attributes that appear on the right-hand side of an assembly invariant). Specifically, we use finite differencing to govern the generation of update methods (e.g., the method that updates the value of `a` given the change in value of `b`) that update a dependent variable when one of the independent variables is modified. Notice that if the invariant expression `E` references `k` different independent variables, then we generate `k` different update methods, each of which re-establishes the invariant in response to a spoiling update of one of the `k` variables.

Data transformers

In our experience generating invariant-maintenance code from applicative invariants, we noticed that constraints over aggregate data types, such as sets and sequences, often require the use of intermediate state to enable efficient updates. Suppose, for example, that a viewport object is used to display a portion of a sequence of text lines. Then some updates to the sequence will not cause any change to the viewport, and it would be nice to eliminate the generation of viewport notifications when they will not cause any change. A *data transformer* is an object that reifies the application of a function over an aggregate data type, such as the derivation of a sub-sequence bounded by two indices, the selection of a maximal element, or a reduction operation over all of the elements in a set or sequence. Data transformers compute and maintain these functions in the face of updates to the subject collection. Moreover, a data transformer can be attached to a subject collection in the object that contains the collection, and thus can be used to minimize the number of notifications required to maintain an invariant. Data transformers are reusable and are a power-

ful abstraction for trading the placement of computation in collaborating components so as to optimize message flow.

Conclusion

A high-assurance system behaves as you expect it to and, most importantly, you know that it does so. The enemy of assurance is complexity, and the main weapons in fighting complexity are abstraction, transparency and intentionality. DYNAMO uses model-based specifications written in OCL to express system properties at a high level of abstraction. Wrapper code is then generated in such a way that each of the specified guarantees are mapped transparently and intentionally into self-contained classes without compromising existing code. Two additional benefits accrue from the DYNAMO approach: flexibility and economy. The code generation architecture and the design of the wrapper code is such that the choice of collaboration mechanism can be made flexibly at assembly time. And the generated code avoids much of the costly indirection common in alternative invariant-maintenance mechanisms.

The DYNAMO approach is one of invariant maintenance. That is, system properties concerning which assurance is desired are expressed as assembly invariants. An assembly invariant relates aspects of one component with those of others. When the state of the former component changes in such a way that the invariant aspect is altered, dependent components must be notified and the invariant reestablished.

A variety of approaches have been developed for invariant maintenance, and DYNAMO introduces another, called a mode component. Mode components are wrapped components organized into a layered, implicit invocation architecture. The wrapping is such that changes to the state of the underlying component are detected and notification made to dependent components without explicit coupling to those components.

The DYNAMO project has also explored alternative, existing invariant-maintenance mechanisms including encapsulated components, mediators, and distributed, peer-to-peer components. These mechanisms, and mode components, share enough structure that common code-generation approaches can be applied to them, thereby enabling assembly-time configuration. This approach to flexible packaging of components promotes reuse.

DYNAMO code generation makes use of the metaprogramming capabilities of the C++ language and compiler. Specifically, DYNAMO expresses the various invariant maintenance mechanisms as templates that are processed at compile time, rather than run-time. Moreover, the templates are organized as mixin layers thereby reducing the need for expensive dynamic binding. The resulting code provides a low-overhead approach to solving the invariant-maintenance problem.

The DYNAMO project has been primarily concerned with a class of system properties called correctness guarantees. These guarantees provide assurance that the results produced by a system are correct. There are, however, other types of properties that must be explored. To this end, we have done some investigation of synchronization guarantees that determine whether a system collaborates in an intended fashion. To this must be added work on quality-of-service guarantees that describes performance and resource-consumption properties of assemblies of components.

References

[1] Andrei Alexandrescu. *Modern C++ Design.* Addison Wesley, 2001.

[2] D. Batory and S. O'Malley. "The Design and Implementation of Hierarchical Software Systems with Reusable Components." *ACM Transactions on Software Engineering and Methodology*, 1(4):355–398, October 1992.

[3] R. Behrends and R. E. K. Stirewalt. "The Universe Model: An Approach for Improving the Modularity and Reliability of Concurrent Programs." *Proceedings of ACM SIGSOFT Symposium on Foundations of Software Engineering (FSE'2000),* 2000.

[4] Antoine Beugnard, Jean-Marc Jézéquel, Noël Plouzeau and Damien Watkins. "Making Components Contract Aware." *Computer*, 32(7):38-45, July 1999.

[5] Roy H. Campbell and A. Nico Habermann. "The Specification of Process Synchronization by Path Expressions." *Volume 16, Proceedings of International Symposium on Operating Systems,* Berlin, Germany, April 23-25, 1974, E. Gelenbe and Claude Kaiser, editors, *Lecture Notes in Computer Science,* Springer Verlag, pp. 89-102.

[6] R. DeLine. "Avoiding Packaging Mismatch with Flexible Packaging." *Proceedings IEEE International Conference on Software Engineering*, pp. 97–106, 1999.

[7] J. Dingel, D. Garlan, S. Jha, and D. Notkin. "Reasoning about Implicit Invocation." *SIGSOFT'98,* Orlando, Florida, November 1998, pp. 209-221.

[8] Erich Gamma, Richard Helm, Ralph Johnson and John Vlissides. *Design Patterns: Abstraction and Reuse of Object-Oriented Software.* Addison-Wesley, 1995.

[9] David Garlan and Curtis Scott. "Adding Implicit Invocation to Traditional Programming Languages." *International Conference on Software Engineering,* 1993, pp. 447-453.

[10] D. Gelernter and N. Carriero. "Coordination Languages and their Significance." *Communications of the ACM*, 35(2):97–107, February 1992.

[11] J. Gosling and F. Yellin. *The Java Application Programming Interface, Volume 2: Window Toolkit and Applets*. Addison-Wesley, 1996.

[12] International Organization for Standardization. "Information Technology—Open Distributed Processing—Interface Definition Language." ISO/IEC 14750:1999, http://www.iso.org/iso/en/CatalogueDetailPage.CatalogueDetail?CSNUMBER=25486.

[13] Corinne McNeely, Spencer Rugaber, Kurt Stirewalt, and David Zook. "DYNAMO Design Guidebook." Technical Report GIT-CC-02-37, College of Computing, Georgia Institute of Technology, June 27, 2002.

[14] Robin Milner. *A Calculus of Communicating Systems.* Springer-Verlag, 1980.

[15] Object Management Group. "Unified Modeling Language, Version 1.4." OMG Document Number 01-09-67, Chapter 6, http://www.omg.org/cgi-bin/apps/doc?formal/01-09-67.pdf.

[16] Robert Paige and Shaye Koenig. "Finite Differencing of Computable Expressions." *ACM Transactions on Programming Languages and Systems,* 4(3):402-454, July 1982.

[17] T. Reenskaug. *Working with Objects: The OOram Software Engineering Method.* Manning, 1995.

[18] Dirk Riehle. "The Event Notification Pattern—Integrating Implicit Invocation with Object Orientation." *Theory and Practice of Object Systems,* 2(1):43-52, 1996.

[19] Spencer Rugaber and Kurt Stirewalt. “Metaprogramming Compilation of Invariant Maintenance Wrappers from OCL Constraints.” Technical Report GIT-CC-03-46, College of Computing, Georgia Institute of Technology, October 28, 2003.

[20] Y. Smaragdakis and D. Batory. “Implementing Layered Designs with Mixin Layers.” *Proceedings of the 12th European Conference on Object-oriented Programming,* 1998.

[21] Mary Shaw and David Garlan. *Software Architecture / Perspectives on an Emerging Discipline.* Prentice Hall, 1996.

[22] Kurt Stirewalt and Spencer Rugaber. “The Model Composition Problem in User Interface Generation.” *Automated Software Engineering,* 7(2):101-124.

[23] K. Sullivan and D. Notkin. “Reconciling Environment Integration and Software Evolution.” *ACM Transactions on Software Engineering and Methodology*, 1(3):229–268, July 1992.

[24] M. VanHilst and D. Notkin. “Using Role Components to Implement Collaboration-Based Designs.” *Proceedings of OOPSLA 1996,* pp. 359–369, 1996.

[25] Jos Warmer and Anneke Kleppe. *The Object Constraint Language.* Addison Wesley, 1999.

Glossary

- *actor:* A participant in the environment is which an assembly lives. Actors can be passive repositories of data or can proactively communicate with the assembly.
- *assembly:* A collection of software components that comprise a system.
- *collaboration:* A protocol of interaction among multiple objects (called *roles*) to achieve some purpose or goal or to implement some invariant.
- *component:* A unit of software assembled with other components to create a larger system.
- *data transformer:* An object that reifies the application of a function over an aggregate data type.
- *distributed integration:* Invariant maintenance accomplished by distributing responsibility amongst collaborating components.
- *encapsulated integration:* Invariant maintenance provided by a single encompassing component that aggregates the other collaborating components.
- *event:* An atomic unit of communications, which may carry data.
- *explicit invocation:* A collaboration mechanism in which (in an OO implementation) an integrand stores references to other integrands and directly invokes operations through these references.
- *finite differencing:* A program optimization that systematically replaces expensive computations of an applicative expression with an explicit variable that maintains the value of the expression and additional code that updates the variable (without recomputing the expression) when one of the dependent variables changes.
- *flexibility:* The property of a system or component that reflects the extent to which the system or component can be used in a variety of configurations.
- *flexible packaging:* A design strategy in which the choice of component collaboration mechanism is delayed until assembly time.
- *guarantee:* A description of expected assembly behavior.
- *implicit invocation:* An architectural style in which components whose status change notify dependent components that have expressed their interest. The notifying component is not explicitly aware of the identity of the notified component.
- *integrand:* a component of a collaboration.
- *intentionality:* The property of a feature that reflects the extent to which the correctness of the feature is readily apparent from the implementation of a system incorporating it.
- *invariant:* A property of an assembly or a component that holds between event occurrences. Invariants are expressed in terms of relationships among the assembly's percepts or status variables.
- *invariant maintenance:* The process by which a system responds to a change in one component in order to reestablish its invariants.
- *mediated integration:* Invariant maintenance provided by a mediator.

- *mediator:* A software component that reifies integration relationships into a component different from the components being integrated.
- *metaprogramming:* An implementation technique for specifying software properties to a compiler and allowing it to generate source code rather than coding the properties directly.
- *mixin class:* Class templates in which the template parameter is used to define the base class from which the (instantiated) mixin class derives.
- *mixin layer:* A collaboration implemented by a template class with one or more nested classes, each of which is a mixin class that defines a different role in the collaboration.
- *mode component:* A hierarchical software component whose interface provides a continuously updated view of its current status.
- *mode component architecture:* A layered, implicit-invocation architecture where all associated components are mode components.
- *mode component constraint:* A single-assignment OCL expression where the left-hand side contains a single status variable and the right hand side is a formula dependent upon status variables of mode components in an adjacent layer.
- *OCL:* The Object Constraint Language. A notation for expressing invariants and pre and post condition information in UML diagrams.
- *OCL constraint:* An OCL expression attached to either a UML class or an association.
- *overhead:* The property of a feature that reflects the performance penalty paid by a system when the feature is added.
- *packaging:* The replacement of abstract communication primitives in a component's source code with actual code that implements the primitives.
- *percept:* Visual feedback; generally an attribute of the assembly that is visible to the user.
- *pre/post conditions:* A specific type of OCL expressions that specify the behavior of component services.
- *response:* A property of an assembly or a component that holds as the result of the assembly or component processing an event. Responses are expressed in terms of relationships among the elements of the assembly's or component's state.
- *role:* A fragment that comprises the subset of an actual object’s characteristics that are required for the object to participate in a particular collaboration.
- *service:* An explicitly invoked operation used by a component in an adjacent layer to alter a component's state.
- *status:* That part of a component’s state that is visible to other components.
- *status variable:* A component attribute that provides automatic notification when its state changes to client components in an adjacent layer.
- *transparency:* The property of a feature of a system or component that reflects the extent the feature can be added to the system or component without requiring alterations to them.
- *wrapper:* A code fragment that can be used to adapt an existing component for a variety of purposes such as providing a different interface or enhancing functionality.