Automated Invariant Maintenance via OCL Compilation

Kurt Stirewalt

Computer Science and Engineering
Michigan State University

Spencer Rugaber

College of Computing
Georgia Institute of Technology

Abstract. UML design models, specifically their declarative OCL invariants, must be refined into delivered code. A key problem is the need to integrate this logic with programmer-written code in a non-intrusive way. We recently developed an approach, called *mode components*, for compiling OCL constraints into modules that implement logic for transparently maintaining these constraints at run time. Specifically, mode components are implemented as nested C++ class template instantiations. The approach makes use of a key device—status variables. The attributes of a component to which other components are sensitive are called its *status*. A *status variable* is a lightweight wrapper on a status attribute that detects changes to its value and transparently invokes a method to handle announcements to dependent components. A mode component is a wrapped code unit containing one or more status variables. The contribution of this paper is a technique for achieving this integration using metaprogramming techniques.

1 Problem Statement

Component-based software development attempts to gain productivity and quality benefits by making use of existing code resources. But even if the existing components are themselves reliable, the resulting assembly might not be. We would like to find ways to improve our confidence in the assembly, while retaining the leveraging benefits. Assume that we start with a specified set of behavioral guarantees, called *invariants,* for the target system. Our quality goal is ensure that the invariants are maintained throughout execution. Moreover, we want to achieve this goal while satisfying the following additional, non-functional properties.

- **Transparency:** The solution should refrain from intruding into the components themselves. Transparency separates reasoning about invariants from the details of the components' implementations. Also, it reduces the need to modify the code of the components, thereby lessening the risk of introducing defects.
- **Flexibility:** There are a variety of architectural approaches for combining components. A flexible solution is one in which an architectural approach can be se-

lected by the designer based on other desirable system properties. Moreover, flexibility supports reuse, enabling components to be packaged in various ways.

- **Economy:** A goal of the composition process is to avoid additional run-time costs over an *ad hoc* implementation. As a general rule, the more encapsulated and self-contained the components are, the more complex is the composition mechanism required to integrate them. With complexity comes run-time overhead. An economical solution supports collaboration without additional run-time cost.
- **Intentionality:** In order to reason about system behavior, it should be possible to relate the behavioral specification of a desired invariant to its implementation directly. In particular, each invariant should be traceable to the code mechanism responsible for guaranteeing it. Intentionality also supports maintainability—changes to system functional requirements often mean altering system invariants. Invariants implemented intentionally are easier to alter.

This paper describes a mechanism for assembling components into a system whose behavior is guaranteed. The composition and its invariant properties are specified by a designer using a subset of UML and OCL. The specified model is automatically compiled into a set of wrappers that enforce the desired invariant properties. The wrappers make use of the metaprogramming features of C++ to achieve the non-functional goals of transparency, flexibility, economy, and intentionality.

2 Solution Approach

2.1 Modeling

The component assembly process described in this paper is called DYNAMO, short for Dynamic Assembly from Models. 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 the Unified Modelling Language[1] (UML) [12] including the Object Constraint Language (OCL) [21]. Moreover, we have interpreted UML class model constructs in terms of the vocabulary of software architecture[2]. (See Table 1.) Annotations to the class model, in the form of OCL constraints, provide semantics. In particular, handlers for external system events (*stimuli*) are ultimately modelled as methods in a component. OCL pre- and post- condition constraints specify the effect of events on the system. Invariants, initially indicated with natural language annotations, are first translated by the designer into OCL annotations to associations. (The UML rule restricting invariants to classifiers is relaxed for this step only.) As the architecture is refined, associations are subsumed by DYNAMO's layered architecture. At this point, each constraint is assigned to the component responsible for maintaining it.

2.2 Design Method

A designer using the DYNAMO method constructs a declarative model of the assembly

1. Specifically, UML v1.4.
2. See [11] for a discussion of the use of UML for modelling software architecture.

Table 1: DYNAMO UML Interpretation

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

expressed using a graphical UML CASE tool. The DYNAMO design method comprises three phases that refine a conceptual model of a proposed assembly into interrelated components organized into layers. In Phase 0, the environment in which the assembly executes is described in terms of external actors, the assembly itself, and the behavioral properties that the assembly guarantees to maintain. Phase 1 partitions the assembly into its constituent components, assigning responsibility for handling external stimuli and invariant-maintenance to the components appropriately. Finally, Phase 2 asks the designer to layer the constituents, where lower-level components communicate status changes upward, and higher-level components make specific service requests of lower-level components. For more details of the DYNAMO design process and a complete elaboration of an example, refer to [10].

2.3 Architecture

In DYNAMO, desired system properties are expressed as invariants using OCL. When an external stimulus perturbs the state of the system, invariants must be re-established. We also wish the process to satisfy the non-functional constraints (transparency, flexibility, economy, and intentionality) described above. We call this process *invariant maintenance*. DYNAMO addresses the invariant-maintenance problem by compiling the OCL invariants into wrappers that transparently notify dependent components when they need to take action to re-establish an invariant. In particular, DYNAMO components are organized into a layered, implicit-invocation architecture. The order of layers is determined by the navigation paths occurring in the OCL constraints, thereby improving intentionally. Implicit invocation, because it is provided by wrappers, enhances transparency. Both improvements add to flexibility and reusability. The implementation approach described in the next section addresses the issue of economy.

A DYNAMO design comprises a layered set of components. For each component, event-handling methods, percepts[3], and OCL constraints are identified. The compilation process takes these three elements as input and produces wrapper code as output. At run-time, the wrappers detect and propagate events and update dependent components, thereby maintaining system invariants.

3 Metaprogramming Implementation

DYNAMO implementation takes advantage of the metaprogramming features of C++.

3. A *percept* is a unit of presentation that communicates system state to the end user.

Specifically, component wrappers are implemented as layered C++ class template instantiations. A class template is a parameterized class definition, where the parameter is usually another class. Moreover, the parameter can be used as the base class of the template class thereby enabling components to be stacked into layers. When combined with C++'s compile-time inlining mechanism, much run-time overhead can be avoided. This section describes how OCL constraints are realized as generated C++ wrappers. To do so, DYNAMO makes use of two devices—status variables and mode components.

3.1 Status Variables

A key concept in our approach to solving the invariant-maintenance problem is that of a status variable. The attributes of a component to which other components are sensitive are called its *status*. A *status variable* is a lightweight wrapper on a status attribute that detects changes to its value and announces them to dependent components.

```
class B {
  protected:
    int b;
  public:
    ...
    void tweak(const int& x) {
      b = x;
    }
    ...
};
```

Figure 1 Schematic class template for an independent component

To illustrate how status variables work, consider the trivial example of two components, A and B, with integer status attributes a and b, respectively, such that variable a must hold exactly twice the value of variable b, regardless of how b changes. That is, there is an invariant between A and B such that `a = 2 * b`. Expressed in OCL, this invariant is `{context A inv: a = 2 * B.b}`. It is assumed that the value of b can change in arbitrary ways. Hence, a C++ schematic for component B is shown in **Figure 1**, where `tweak` is an arbitrary method representative of the various ways in which the value of b might be altered. When `tweak` is called, b's status changes, thereby requiring an update to a. A solution to the invariant-maintenance problem requires a means of updating component A whenever `tweak` is invoked.

We implement a status variable's update behavior by wrapping the definition of the variable's class with a *listening agent*, such as is described in [18], that exports the same abstract interface as the existing class. To do this, status variables take advantage of several C++ features, including its ability to overload the assignment operator. That is, when an overloaded assignment is made to a C++ variable, a programmer-provided method is invoked to perform additional activities. The power of status variables is their use of assignment overload to transparently detect changes of status.

Each status variable has its own class that is produced by instantiating the class template `StatusVariable<T>` shown in **Figure 2**. The template parameter T is the type of the attribute to be wrapped. In the case of attribute b, the type is `int`. Status variable classes have one attribute of their own, named `data` (line 9), protected from external access. This attribute holds the actual value being wrapped. Changes to b are trapped by the assignment overload method (`operator=`) on line 6. This method is `virtual` (polymorphic) and will be extended in the derived class by a method that notifies com-

```
( 1) template <typename T>
( 2) class StatusVariable {
( 3)   public:
( 4)     StatusVariable() {}
( 5)     StatusVariable(const T& t) : data(t) {}
( 6)     virtual T& operator= (const T& t) {data = t;}
( 7)     virtual operator T() {return data;}
( 8)   protected:
( 9)     T data;
(10) };
```

Figure 2 StatusVariable class

ponent A that b has been altered. The only responsibility that the assignment overload operator has in the `StatusVariable` class is to assign the new value to `data`.

Clients of status variables, such as component A, do not know that attribute b has been wrapped. Hence, when they request the value of b, they must be provided an `int`, not a `StatusVariable<int>`. C++ provides a supporting mechanism, called a user-defined conversion, as illustrated on line 7 by `operator T()`. In the example, T is `int`, and the `int()` method is invoked whenever the value of b is requested, either explicitly within the code of B, or implicitly, via compiler-generated conversions. Hence, the `int` value of `data` is returned whenever the value of the status variable wrapping b is requested. The `StatusVariable<T>` class also provides constructors (lines 4 and 5) useful both for initially establishing invariants or in case class B provides an externally visible way to initialize b.

3.2 Using Status Variables

Given a constraint, its dependent and independent variables can be determined[4]. Changes to the independent variables must be detected and the associated dependent variables adjusted to reflect the change. That is, each independent variable in each constraint must be wrapped as an instance of a class derived from `StatusVariable<T>`. The name of the class is formed from the name of the status variable and the component containing it, thereby ensuring uniqueness. For variable b of component B, the generated template class has the name SV_B_b. SV_B_b has the form illustrated in **Figure 3**.

Note that SV_B_b derives from `StatusVariable` (line 2) and overrides the assignment operator (lines 9-13). The override invokes the assignment operator in `StatusVariable`, thereby storing the assigned value. It then invokes an update method (`update1`). The update method, which also must be generated, lives in component A, as wrapped, and contains the code to retrieve the new value of b and update a accordingly. When SV_B_b is generated, it must know the name of the update method (`update1`) and which component it lives in (A). It obtains this information when the `setUpdater1` method (lines 6-8) is called by the component containing the status variable (B, as wrapped).

4. There are some *non-constructive* constraints for which this may not be possible. They are discussed in section 4.5.

```
template <typename T>
class SV_B_b : public StatusVariable<T> {
  public:
    SV_B_b() {}
    SV_B_b(const T& x) : StatusVariable<T>(x) {}
    void setUpdater1(Updaters* sc1P) {
      updater1P = sc1P;
    }
    T& operator=(const T& d) {
      StatusVariable<T>::operator=(d);
      if (updater1P)
        updater1P->update1();
    }
  protected:
    Updaters* updater1P;
};
```

Figure 3 Status change announcement mechanism

3.3 Mode Components

It remains to describe how dependent components (such as `A`) are bound to independent components (such as `B`). In the example, `A` is responsible for updating the value of `a` when `b` is changed. It does this in a generated method, `update1` (line 6 of **Figure 5**). That is, a new method for `A` (`update1`) is generated, which is called when `b` changes. Its responsibility is to request the new value of `b` and, using it, to recompute the value of `a`. This raises several questions: Where does the code for `update1` live? How does `b` know to call `update1`? And how does `A` know how to obtain the value of `b`?

The `update1` method logically lives in component `A`. However, as we wish to leave existing components untouched to the extent possible, we generate a new wrapper that extends `A` with the update method. The other two questions can be resolved by further wrapping `B` in such a way that the required information is available. Once `B` is wrapped, it becomes a mode component. A *mode component* is a wrapped component containing one or more status variables. The mode component wrapper for `B` is named `B_Top` (shown in **Figure 4**), and it is generated based on the status variables and invariants specified for the assembly[5].

`B_Top` is a class template. Moreover, it is a *mixin* class template [2]. This means that its template parameter is a class, and that `B_Top` derives from that class. That is, `B_Top` is a subclass of the class bound to the template parameter `T`. Mixins are used as a way to provide behavior to a class in addition to that derived from its normal base

5. Note that the `_Top` and `_Bot` suffixes on template class names refer to their roles in the layered architecture and not to their roles in the inheritance hierarchy. That is, the `_Top` wrapper provides services that communicate with a component above it in the layered architecture. The relative nesting of the templates is actually in the inverse order to their position in the layering.

```
( 1) template <typename T>
( 2) class B_Top : public T {
( 3)   public:
( 4)     B_Top() {};
( 5)     B_Top(const int& x) : T(x) {}
( 6)     int getValue_b(void) {return(b);}
( 7)     void bind_b_1(Updaters* scP) {
( 8)       b.setUpdater1(scP);
( 9)     }
(10) };
```

Figure 4 Mode component wrapper for component B

class. In the case of B_Top, its parameter is B. If A then refers to B_Top instead of B, it will obtain the extended behavior.

B_Top adds two methods to those available in B. Method getValue_b provides access to the status variable b's value. It can be called by A when A is alerted to changes in b. Method bind_b_1 illustrates the mechanism whereby A can inform B of any invariant re-establishment methods that must be called when B's status changes. Specifically, bind_b_1 is the means by which changes to b are communicated in order to maintain the first invariant (1). Its argument is a pointer to the update method in A (update1) responsible for maintaining the invariant. bind_b_1's responsibility is to communicate this pointer to the status-variable wrapper for b (line 8 of **Figure 4**).

The binding between components related by invariants is complex. Dependent components like A must be able to request status variable values, such as b. To do this, A must have access to B, the component that contains b. A straightforward way to do this is to have A contain a pointer to B. But pointers are costly, each access requiring the dereferencing of the pointer. The C++ template mechanism can sometimes avoid this overhead by having A derive from B as a mixin. Then A can have direct access to b, just like it can to its own instance variables.

To summarize: A has four responsibilities that arise due to its interaction with B: 1) It must derive from B in order to access it efficiently; 2) it must let B know how to alert it when changes occur; 3) once alerted, it must access the value of b; and 4) it must re-establish the invariant by recomputing the value of a.

To discharge these responsibilities while maintaining transparency, another wrapper is used (**Figure 5**). A_Bot is a mixin class template. Its template parameter is the component upon which it is dependent, B (as wrapped by B_Top). A_Bot mixes B in via private inheritance, thereby hiding B from subsequent classes derived from A. This inheritance discharges responsibility 1. In addition, A_Bot inherits publicly from two other classes, A and Updaters. Updaters is an interface class containing declarations for the types of updater methods.

The key feature of A_Bot is the update1 method on line 6. This is the method called by the status variable b when it detects a change to its own value. Notice that update1 accesses the value of b by using the getValue_b member function of

```
(1) template <typename T>
(2) class A_Bot : public A,
(3)   public Updaters, private T {
(4)   public :
(5)     A_Bot() {myB.bind_b_1(this);}
(6)     void update1() {a = 2 * myB.getValue_b();}
(7)   protected :
(8)     T myB;
(9) };
```

Figure 5 Wrapping dependent components

component B. This method discharges responsibility 3. Line 6 also illustrates how the invariant is re-established to discharge responsibility 4.

Responsibility 2 is handled by the wrapper's constructor shown on line 5. When component A is instantiated, the method `bind_b_1` is called in component B, passing the address of component A itself as an argument. The address is passed in turn to the `setUpdater1` method of SV_B_b, where it is stored for use when b changes value.

Putting the pieces of the example together requires a nested template instantiation, such as `A_Bot<B_Top> myA;` which declares an assembly myA as the composition of A (as wrapped) with B as wrapped. Notice that stacking components in this fashion easily generalizes. If B itself was dependent on a status variable in component C, another level of nesting could be used.

The overall mode-component architecture is presented in **Figure 6**[6]. `StatusVariable<int>` contains space for the actual value being monitored and provides default operations for assignment override and type conversion. Actual status variable classes, such as `SV_B_b<int>`, override the assignment operation to invoke any listeners, such as `update1`. `Updater` is an abstract class containing pure virtual methods for each of the constraint update methods. B and A are the original components containing, respectively, attributes b and a. They both must be wrapped in order to become mode components. Because A contains a dependent status variable, a, it is above B in the component layering. Its wrapper, `A_Bot`, must therefore provide a downward-looking service, `update1`, for updating status variable a. Conversely, B's wrapper, `B_Top`, must provide upward looking services, such as `getValue_b` and `bind_b_1`. `GetValue_b` enables A to retrieve the updated value of b; `bind_b_1` provides a way for letting B know which update service in A to invoke.

3.4 Fine Print

In order to clearly explain status variables and mode components, several details of the invariant maintenance process have been glossed over in the description above. Foremost among them is the seeming separation of A's invariant re-establishment wrapper (`A_Bot`) from B's announcement wrapper (`B_Top`). In reality, A itself may contain independent status variables participating in other invariants. For example, component Z

6. To simplify the diagram, the template classes themselves and the corresponding «`bind`» dependencies are not shown.

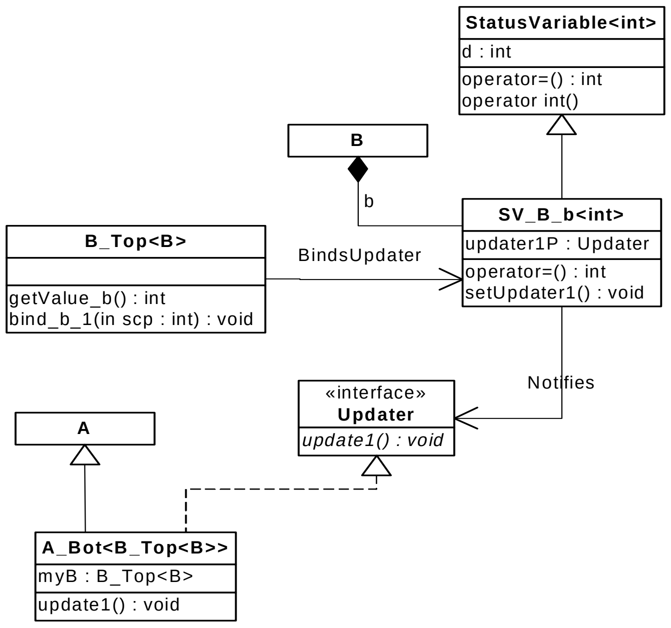

Figure 6 Mode component implementation architecture

might depend on variable a of component A. This would imply the need to generate an A_Top wrapper similar to B_Top. Because of the nesting enabled by C++ templates, both A_Top and A_Bot can be used to wrap A.

Other details not discussed are status variable initialization and the initial establishment of any relevant invariants. If, for example, component B provides a way to initialize the value of variable b, then the generated code has to include a memberwise initializer for it that incorporates a call to any relevant updaters. The actual compilation process also includes generating several include files providing access to required names.

Another issue concerns OCL collection classes. The example elaborated on in this section does not make use of any of collections. Actually, collection classes themselves are just another form of value that can serve either a dependent or independent role in an invariant. But we do not want a change to a single element of a collection to alert all components dependent on the collection, but only those dependent on the altered element. C++ template nesting can help address this issue as well. We have experimented with inserting intermediate template class instantiations, called *data transformers*, that can optimize certain invariant-reestablishment operations on collections by intercepting and mediating the corresponding update requests.

The example also made only fleeting use of OCL navigation. In actual practice, OCL constraints can included a cascade of classifier names to relate topologically distant components. Navigation such as this can be handled in DYNAMO by using C++'s name scoping operator (::) to directly access variables in nested components.

3.5 Extending the Example

The approach described above illustrated how an invariant dependent on a single status variable can be maintained. Real systems are more complex. This section describes how the example can be generalized.

Multiple status variables. For each status variable `x` of type T_x aggregated by a component `K`, there is a corresponding generated class `SV_K_x`. Each `x` must be defined within `K` as a normal instance variable, but with type `SV_K_x<`T_x`>`. There is no limit to how many such variables `K` can have. Note that it is the responsibility of each component to maintain its own intracomponent invariants.

Multiple constraints. A given independent status variable, `x`, belonging to component `K`, may be involved in multiple constraints (C_i). Hence, multiple updates may have to be performed when the value of `x` changes. For each such constraint, an update method (updateC$_i$) and a bind method (bind_x_C$_i$) must be generated. Moreover, the code in the `SV_K_x` class must invoke each of the updater methods (updateC$_i$). Finally, the addresses of the updater methods must be remembered in the `SV_K_x` class with function pointers (updaterC$_i$P).

Circularities. In the example above, component `A` is notified of changes to component `B` and then requests new values from it. The mode component mechanism for accomplishing this takes advantage of the C++ ability to nest templates. That is, component `A` as wrapped has as a template parameter component `B` as wrapped. This mechanism is inherently asymmetric. That is, it cannot be used to have component `A` notify component `B` because of the resultant circularity in the template instantiation ordering.

Several things should be noted about a circular dependency such as this. First, there is no reason why components `A` and `B` cannot use traditional intercomponent messaging when `A` needs to notify `B` of a change. That is, `B` can provide an update method that `A` can call directly. The second observation is that a circularity is often a symptom of a design problem. One manifestation of the problem is an endless loop—`B` notifying `A` which notifies `B`, repeatedly. Hence, any circularity in the dependency graph may be a sign of a design problem and should be carefully examined.

Multiple components per layer. Sometimes circular dependencies are inherent but do not lead to an endless loop. This can occur when a status variable (`p`) in one component depends on a status variable (`q`) in another, and `q` depends on a different status variable (`r`) in the first. While this situation is circular as far as template nesting is concerned, it does not lead to infinite update when one of the variables is changed. As an alternative to the asymmetric mechanism of mode components, both components can be configured as nested classes contained within a single mode component class, such as with mixin layers [16].

3.6 Tool Support

DYNAMO designs are expressed using an OCL-capable UML modeling tool such as Rational/IBM [8] or ArgoUML [19]. These tools support the export of diagram content and associated OCL annotations in the industry-standard XMI CASE-data interchange format [13]. We have written tools for extracting relevant information from XMI, representing it in a target-independent abstract syntax tree (AST) and generating code from the AST. Code generation consists of two steps: conversion to an internal representation (IR) and traversal of the IR to generate C++ wrapper templates. Further details concerning the compilation process can be found in reference [14].

4 Evaluation

4.1 Transparency

What alterations to the source code of existing components are required in order to make them into mode components? Only one change is necessary on the part of a programmer—the types of status variables must be adjusted. That is, member variables of components upon which other components are dependent must be so designated. Two scenarios can be imagined. In the first, the original designer of a component library is seriously concerned with reuse. Components are developed, and potentially interesting status is declared as such in the component code. The second scenario is the adaptation of an existing component into a mode component. In this case, the adaptor must not only decide what facilities of the component are required of other components, but must also locate the definitions of these variables in the code, so that their types may be altered. In both scenarios, the coding effort required of the developer consists of adding some `#include` statements and changing the types of the status variable declarations. Any scheme for intercomponent invariant maintenance must provide access to the constituent state. Hence, we judge the mode component approach to be adequately transparent.

4.2 Flexibility

The DYNAMO approach is flexible in several senses. First is the fact that alternative components with the same APIs can be substituted for each other. Moreover, additional component can be inserted to provide optimizations and other enhancements. These added or substituted components simply amount to interpolated templates in the C++ code. DYNAMO is also flexible in a different sense. Mode components are not the only scheme for maintaining invariants. For example, mediators [17] provide many of the same features. More conventional approaches to invariant maintenance in C++, such as aggregated components with embedded pointers and explicit delegation can also be used. The DYNAMO compilation architecture has been successfully applied to these alternative approaches. That is, the DYNAMO compilation approach is flexible with respect to the specific mechanism for updating status to maintain invariants.

4.3 Economy

Flexibility normally leads to overhead. Typically, flexibility is achieved by using indirection through pointers. Using pointers implies dereferencing, which, in turn, means an extra operation on every access. Our approach reduces overhead by making use of two features of C++: template classes and inlining.

Components are normally constructed independently and encapsulated in their own classes. This reduces coupling and enhances maintainability. But, because components need to interact, they often hold pointers to each other. Another approach is to have one component be a subclass of another. Then the subordinate can directly access the features of the superordinate component without the pointer overhead. But such an approach is intrusive and unnatural. Mixin inheritance is an alternative to subtyping—a mixin adds a feature to a class without requiring that the mixin be an explicit subtype.

The other C++ feature that can reduce overhead is *inlining*. Normally, the compilation of a method call introduces significant overhead at the calling site. The C++ compiler can detect situations where a copy of the code for the called method can be inserted directly at the call site without the associated overhead. This technique is particularly applicable when the method code is short, such as obtains with instance-variable access routines (getters and setters). In this way, components can retain their encapsulation without engendering normal intercomponent communication overhead. Templates and inlining enable our approach to provide low overhead invariant maintenance.

4.4 Intentionality

The overarching goal of the DYNAMO work on component assembly is to increase assurance. It accomplishes this by providing an invariant-maintenance mechanism. Invariants are directly manifest in the code. In particular, each independent variable in each invariant results in the generation of a status-variable wrapper to provide change notification and an update method to re-establish the invariant when one of its constituents changes. Because this code is generated, it is possible for the designer to have confidence that the specification is being met. Hence, the approach is intentional[7].

4.5 Limitations

The DYNAMO approach, while satisfying the above-described non-functional goals, is not without limitations. Some of these are described here.

- **Loss of symmetry:** Components nested as template mixins are inherently asymmetric. This loss of flexibility is compensated for by the reduced overhead they require.
- **Constructiveness:** Not every invariant can be expressed as a mode component constraint. Constraints in which a single variable appears on the left hand side[8] are called *constructive*. This is a theoretical limitation of the approach that has not proven a problem in practice.
- **Circularities:** More serious are cyclically dependent constraints, as for example, happens if variable a depends on variable b in one constraint, and variable b depends on variable a in another. Run-time update of one variable can lead to an infinite cascade of invariant re-establishments. In DYNAMO, such co-dependencies can be grouped into the same mixin layer, providing a symmetric solution.

7. *Intentional Programming* [4] is an alternative metaprogramming approach that provides intentionality.

8. Some constraints may be algebraically manipulated to solve for a target independent variable.

- **Code obfuscation:** The DYNAMO metaprogramming approach generates C++. Several difficulties arise if this code needs to be maintained. First, the generated code comprises deeply nested class templates; reading and understanding it requires in-depth knowledge of C++. Also, should the code ever have to be edited and recompiled, any ensuing compiler error messages will be hard to interpret.

5 Related Work

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 its integrands). Under this approach, the integration component might be a peer of its integrands, as is the case with mediators [17], or it might encapsulate its integrands, as with GenVoca layers [1]. 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 [7]. At the other extreme, an invariant can be implemented as a collaboration [20], which distribute the responsibilities for maintaining the invariants among the integrands. An alternative to choosing an invariant maintenance mechanism at the time when the code is written is delaying the decision until assembly time. This has been called the *flexible-packaging* problem, and an approach to providing it is described in [5].

DYNAMO makes use of the template processing mechanism of the C++ compiler to obtain its metaprogramming functions. An alternative approach is provided by the Open C++ project [3]. Open C++ adds the meta-object protocol to the C++ compiler. That is, programmer have the ability to reprogram the compiler by, for example, telling it what to do when it sees a new construct, such as a `MonitoredClass`. This construct might be realized with code that counts method calls or variable updates. The metaprogrammer is responsible for using available features of the Open C++ API to write metaprograms for doing the counting. We have successfully applied this tool to generate DYNAMO status variable updates, so it would seem to provide a viable alternative to the template program approach described in this paper. A survey of other work on invariant maintenance can be found in reference [15].

On the issue of implementation, currently, the most complete OCL compiler comes from the Dresden University of Technology and supports OCL 1.4. To support OCL 2.0, the Dresden development team is redesigning their compiler as described in reference [9]. The Dresden compiler features a MOF (Meta Object Facility) Repository that manages models and meta-models by providing interfaces for their access. The code generator itself is designed to take instances of the OCL metamodel as input and output Java code without altering the state of the environment.

6 Summary and Conclusions

A high-assurance system behaves as you expect it to and, just as 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 invariants 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 are 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, critical system properties 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 a participant in the invariant is altered, dependent components must be notified and the invariant re-established.

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.

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 mixins, thereby reducing the need for indirection. The resulting code provides a low-overhead approach to solving the invariant-maintenance problem.

Acknowledgments

The PIs on this project wish to thank the following student participants: Jonathan Gdalevich, Corinne McNeely, Terry Shikano, Patrick Yaner, and David Zook from Georgia Tech and Reimer Behrends and AliReza Namvar from Michigan State. We also wish to thank colleague Laura Dillon from Michigan State. This effort was sponsored by the Defense Advanced Research Projects Agency, and the United States Air Force Research Laboratory, under agreement number F30602-00-2-0618. Other support was provided by Office of Naval Research grant N00014-01-1-0744 and by NSF grants EIA-0000433 and CCR-9984726.

References

[1] 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.

[2] Gilad Bracha and William Cook. "Mixin-based Inheritance." *Proceedings ECOOP/ OOPSLA '90,* October 21-25, 1990, 303-311.

[3] Shigeru Chiba. OpenC++ Home Page. http://www.csg.is.titech.ac.jp/~chiba/ openc++.html.

[4] Krzysztof Czarnecki and Ulrich W. Eisenecker. “Intentional Programming” Chapter 11 in *Generative Programming.* Addison Wesley, 2000.

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

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

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

[8] International Business Machine Corp. “Rational Software.” http://www-306.ibm.com/software/rational/.

[9] Loecher, Sten and Ocke, Stefan. “A Metamodel-Based OCL-Compiler for UML and MOF.” Department of Computer Science. Dresden University of Technology. September 2003.

[10] 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, ftp://ftp.cc.gatech.edu/pub/coc/tech_reports/2002/GIT-CC-02-37.ps.Z.

[11] N. Medvidovic, D. S. Rosenblum, D. F. Redmiles and J. E. Robbins. “Modeling Software Architectures in UML.” *ACM Transactions on Software Engineering and Methodology,* 11(1):2-57, January, 2002.

[12] 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.

[13] Object Management Group. “XML Metadata Interchange (XMI).” http://www.omg.org/technology/documents/formal/xmi.htm.

[14] 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, http://www.cc.gatech.edu/dynamo/papers/compile.pdf.

[15] Spencer Rugaber and Kurt Stirewalt. “Final Project Report / Dynamic Assembly from Models (DYNAMO)”. Technical Report, GIT-CC-05-03, College of Computing, Georgia Institute of Technology, March 2005, ftp://ftp.cc.gatech.edu/pub/coc/tech_reports/2005/GIT-CC-05-03.pdf.

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

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

[18] R. N. Taylor *et al*. “Chiron-1: A Software Architecture for User Interface Development, Maintenance, and Run-Time Support.” *ACM Transactions on Computer-Human Interaction,* 2(2):105–144, June 1995.

[19] Tigris.org. “Welcome to ArgoUML.” http://argouml.tigris.org/.

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

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