Position Paper: The Tradeoff Between Dependability and Efficiency in Embedded Systems

Spencer Rugaber
College of Computing
Georgia Institute of Technology
Atlanta, Georgia 30332-0280
spencer@cc.gatech.edu

R. E. Kurt Stirewalt
Dept. of Computer Science and Engineering
Michigan State University
East Lansing, MI 48824
stire@cse.msu.edu

ABSTRACT

Efficiency and dependability are antagonistic. The optimizations that we design to improve performance and reduce resource consumption add to the complexity of the systems that we build. And because complex systems are hard to understand, we cannot depend on them to behave as we expect. In this paper we explore the nature of the tradeoff between dependability and efficiency. In particular, we examine two sources of complexity: the structural complexity that arises as we add a component to a system and the communication complexity that obtains as the components coordinate their activities. We then present two related strategies for managing that complexity: a layered architecture that reduces intercomponent connectivity and an implicit-invocation mechanism that removes much of the explicit communication among the components. But because efficiency and dependability are antagonistic, reducing complexity to make our systems more dependable runs the risk of making them less efficient. Hence, we also present an implementation architecture that supports layering and implicit invocation without paying the performance penalty usually associated with these techniques.

Categories and Subject Descriptors

C.3 [**Real-time and Embedded Systems**]; D.1.11 [**Software Architecture**]; D.2 [**Software Engineering**]; D.3.4 [**Code Generation**]

General Terms

embedded systems, software architecture, synthesis

Keywords

code generation, layered architecture, implicit invocation

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. To copy otherwise, to republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee.
CASES 2001 Atlanta, Georgia USA
Copyright 2001 ACM X-XXXXX-XX-X/XX/XX ...$5.00.

1. INTRODUCTION

Managing the tradeoff between dependability and efficiency in embedded systems design is a major concern of the DYNAMO project. The key inhibitor of dependability is complexity. In order to be assured of correct execution, analysts, designers, programmers, and users must be able to accurately anticipate system behavior. One aspect of complexity is structural. That is, the greater the number of components in a system and the more interdependent they are, the more difficult it is to comprehend them. Another aspect of complexity is behavioral. Even two components can interact in complex ways if their interdependence is such as to require intricate coordination. Unfortunately, traditional methods for combating complexity, such as information hiding and abstraction, often are paid for with degraded system performance and increased resource usage, the two central aspects of system efficiency. Hence, balancing these two overriding forces—dependability and efficiency—is a central, perhaps *the* central problem in embedded systems development.

We offer two key insights to address the problem:

1. Structural complexity can be reduced by limiting the allowed interactions among the components of a system. This issue is described in Section 2.1.

2. Behavioral complexity can be reduced by expressing interdependency at a high-level of abstraction. This issue is described in Section 2.2.

These insights, in turn, are supported in DYNAMO by two integrated techniques:

1. A layered, implicit-invocation architecture to reduce complexity, as described in Section 3.

2. A compile-time program composition mechanism that reduces procedure call overhead among system components, as described in Sections 2.3 and 4.

Together, these two techniques provide a comprehensive approach to deal with the dependability–efficiency tradeoff.

2. BACKGROUND

Embedded systems often live in a resource-constrained environment. For example, a key constraint of a cell phone is battery life, and one way of increasing battery life is to turn

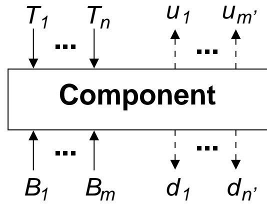

Figure 1: Component in a layered architecture.

down clock speed, thereby compromising performance. Likewise, embedded systems in consumer products face resource constraints, such as limited memory, due to cost competition. Despite these forces, demand has grown for increasingly complex embedded systems. And with increased complexity comes the danger of reduced dependability. Consequently, dealing with the tradeoff between efficiency and dependability has challenged the embedded systems development community. To meet this challenge, DYNAMO applies three existing technologies:

- To reduce structural complexity, DYNAMO makes use of the GenVoca style of layered software architecture.
- To reduce behavioral complexity, DYNAMO uses implicit invocation for some of its inter-component interactions.
- To efficiently implement these two approaches DYNAMO makes use of two advanced C++ features: templates and virtual base classes.

This section provides background information on these three technologies.

2.1 Dealing with structural complexity: Layering

DYNAMO mitigates structural complexity by adopting a layered architecture. In a general software architecture, any component may interact with any other component. In a layered architecture, however, components are grouped into layers in such a way that components in one layer may interact only with components in layers immediately above or below that layer [7]. Hence, the number of possible interactions is significantly reduced, with a corresponding decrease in system complexity. The *interface* of a component is anything that is visible outside the component; everything else about the component is considered part of its *implementation* [6]. Figure 1 illustrates the structure of a component in a layered architecture. In this figure, the T_i are *top* operations, which higher level components can invoke to obtain component services. Conversely, the B_i are *bottom* operations, which lower level components can invoke to obtain component services. The set $\{T_1, \ldots, T_n, B_1, \ldots, B_m\}$ is the component's interface. Similarly, the component's own implementation may make *down* and *up* calls (respectively d_i and u_i) to obtain services from lower (respectively higher) level components. In the figure, solid lines denote calls into the component; whereas dashed lines denote calls outward.

In DYNAMO, we make use of a particular style of layered architecture called GenVoca [3]. In the GenVoca style of layered systems, a layer consists of one or more *parameterized components*. In a parameterized component, parameters specify the type(s) of lower level component(s) that

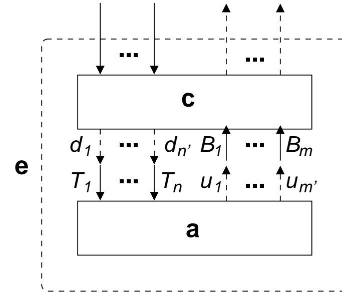

Figure 2: Layered assembly of components ala GenVoca.

the component uses in implementing its services. Component composition, or *layering*, is realized by instantiating one component with other components, such that each actual parameter type checks with the corresponding formal parameter. A component `b` that must be instantiated with components of types T_1 and T_2 is written: `b[T1, T2]`. A particular instantiation of components must satisfy a *type equation*. The type equation

```
e = c[a]
```

represents a component `e` whose top layer is a `c`-component and that implements its services using an `a`-component (the bottom-most layer). Figure 2 illustrates the component assembly that corresponds to this type equation. Observe that down calls in `c` are connected to top operations provided by `a`, and that up-calls in `a` are connected to bottom operations provided by `c`.

2.2 Dealing with behavioral complexity: Implicit invocation

When a system executes, its components interact to accomplish the system's overall goals. Layering can reduce the number of components with which a given component interacts, but it does nothing to reduce the complexity of any specific, pairwise interaction. Pairwise component interaction can be as simple as a one-time invocation followed by a return or as complex as a full-blown communications protocol. Normally, the unit of interaction is the *event*, usually implemented as a procedure call (invocation) which consists of a name and possibly some data values (arguments to the invocation). In general, interaction complexity arises from the sheer number of events and any interdependencies among them.

To reduce behavioral complexity, DYNAMO makes use of an approach called implicit invocation, in which a recipient component is automatically notified when a specific condition arises in a sending component [7]. Architecturally, this reduces the number of interactions that must be explicitly specified. It does, of course, require an underlying implementation to effect the notification. But this implementation approach is both generally applicable throughout the system and highly stylized, allowing the specification to remain abstract and declarative.

2.3 Implementing efficiently: C++ templates and virtual inheritance

DYNAMO aims to mitigate complexity without reducing efficiency. This is accomplished by representing the essen-

```
class Server {
 public:
  void service() { ...  }
};
template<class SERVER>
class Client :  SERVER {
 public:
  void function() { ..SERVER::service();...  }
};
Client<Server> sys;
sys.function();
```

Figure 3: Client–server composition using parameterized inheritance.

tial aspects of layered and implicit-invocation architectures in terms of advanced C++ composition idioms. Specifically, we use two idioms: parameterized inheritance to efficiently implement layered composition, and the multiple-inheritance diamond pattern to efficiently implement event management. We now briefly describe these techniques.

Parameterized inheritance uses the C++ template mechanism to define a class whose parent class is not specified until the template is instantiated. A template is a C++ compile-time mechanism for specifying parameterized functions or classes. Template instantiation binds template formal parameters to the actual values supplied by the programmer. Instantiation creates a corresponding (non-template) function or class which can then be used by the programmer in the usual manner and compiled by the C++ compiler. For example:

```
template<class SUPER>
class Mixin :  public SUPER
 { ...  };
```

defines a class called `Mixin` that may declare instance variables and methods. Moreover, given an arbitrary class `C`, `Mixin<C>` describes a new class that results from binding `C` as the parent class of `Mixin`. Template classes, such as `Mixin`, whose parent class is a template parameter are often referred to as *mixin classes* [10].

Suppose we have a class `Client` whose methods need to invoke the services of another class `Server`. If we construct `Client` and `Server` objects, then for the `Client` object to invoke these services, it would need to call them through a pointer to a particular `Server` object. However, by implementing `Client` as a mixin and instantiating it with `Server`, we get an efficient implementation that does not require such indirection. Figure 3 illustrates this solution. When used with care, parameterized inheritance provides a flexible and efficient separation between classes that *define* the implementation of a set of operations and classes that *use* these operations.

Mixin classes are not the only way to achieve composition of client and server classes. Another technique is to use a variation of multiple inheritance, sometimes referred to as the *diamond pattern* [10]. Multiple inheritance allows a derived class to obtain features from two or more base (parent) classes. If those parent classes have a common ancestor in the inheritance hierarchy, then the relevant classes take the shape of a diamond, as illustrated in Figure 4. The figure uses the UML notation [4], in which rectangles denote

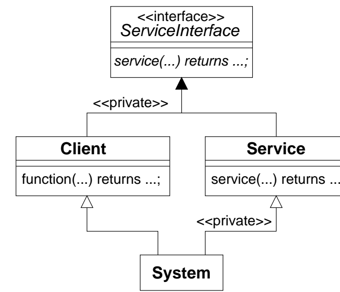

Figure 4: Client–Server composition using the diamond pattern.

classes and lines ending in a triangle denote inheritance relations, with the triangle pointing to the super-class in the relation. In the case where the diamond property holds, the question arises as to whether the child (the bottom of the diamond) inherits multiple copies of the features of the ancestor class (at the top of the diamond). In C++, if the keyword **virtual** is used[1], a single copy of the ancestor features is included in the child. A side effect of the diamond property is that references to ancestor features in the parents are also shared. For example, if the ancestor feature is a pure virtual method, such as service in Figure 4, and if one parent (Server) provides an actual implementation for it, then the other parent (Client) can access the implementation in its sibling, providing a kind of side-ways access. Generally speaking, when this pattern is used, the class that plays the client role will inherit the virtual base class using *private derivation*, so that the operations in that base class are visible to methods in the client class but will not become part of the client's interface. In UML, the keyword private denotes private inheritance.

Figure 4 illustrates the client–server example implemented using multiple inheritance. In this figure, ServiceInterface is an abstract class that defines an abstract (i.e., pure virtual) operation service. Class Service is a concrete class that inherits from ServiceInterface and provides the implementation method for the service operation. Class Client is a concrete class that define an operation function, implemented by invoking the method service(). Methods in Client may invoke operations defined in ServiceInterface by name, as Client inherits the signatures for all of these operations from ServiceInterface. Observe also that, because Client uses private inheritance, operations in ServiceInterface do not become part of the Client interface. Finally, class System uses multiple inheritance to combine classes Client and Service, thereby binding the uses of the service operation in Client to the method that implements it, which is defined in Service.

3. ARCHITECTURAL APPROACH

DYNAMO addresses the dependability–efficiency tradeoff in the context of reactive embedded systems. Specifically, we are investigating how to implement a layered implicit-invocation architecture in a general and efficient way. The key insight is that we can assemble a collection of components in this architecture using a highly-factored design structure that enables:

[1]In UML, virtual base-class inheritance is denoted by a filled triangle.

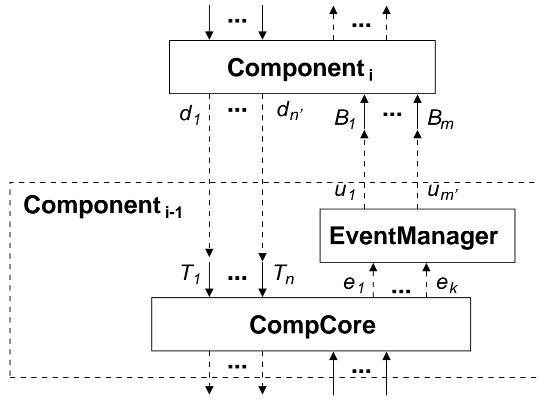

Figure 5: Assembly architecture: layering + implicit invocation

1. Structural dependability afforded by GenVoca's style of layered assembly. Using a layered architecture increases abstraction by reducing component interaction. Moreover, the GenVoca style of layered architecture encourages both dependability, via its type checking, and efficiency, by its mixin implementation of parameterized components.

2. Behavioral dependability afforded by the use of implicit invocation. Implicit invocation likewise raises abstraction and reduces complexity by simplifying interaction protocols. And a templatized implementation enables implicit invocation with minimal procedure-call overhead.

An *assembly* in DYNAMO consists of a sequence of layers stacked on top of each other. Events are passed upward, and service requests are made downward. Each layer consists of one or more components that react to events from below, announce events upward, and make service requests downward. Normal, peer-to-peer interaction is also available among components in the same layer. A component has two parts: a computational class, responsible for providing the services of the component and an event manager responsible for binding events from below to bottom services in the component. Event managers are introduced as a conceptual vehicle to support upward event notification, but they are optimized away in the implementation as described in Sections 4 and 5. Utility classes, unassociated with events, may also reside in a layer, providing support for the layer's components.

Figure 5 illustrates how DYNAMO components are assembled. This figure depicts two distinct components. The lower component is expanded to show its computational-core (CompCore) class and event manager (EventManager). Because $\mathsf{Component}_i$ sits at a higher level in the assembly, its down-calls are explicitly bound to the top services of $\mathsf{Component}_{i-1}$. Computational-core classes in the assembly do not contain explicit up-calls. Rather they provide information to higher-level layers by announcing events. The events announced by the computational-core class in $\mathsf{Component}_{i-1}$ are bound to bottom services of $\mathsf{Component}_i$ by the event manager. The conceptual indirection provided by the event manager allows designers to develop a component's computational-core class without worrying about the details of the layers that might require the services of a component, thereby mitigating one source of behavioral complexity.

Shaw and Garlan [7] discuss different implementations of implicit invocation, from which they tease out several key design decisions, each of which has consequences on the flexibility and efficiency of the resultant systems. DYNAMO focuses on two of these design decisions: the mechanism by which events announced in a lower-level component are bound to bottom services in a higher-level component and the mechanism by which a component announces events. Events from one component may be bound to procedures in another in one of two ways: statically or dynamically[2]. In static binding, events are bound to procedures at compile time; whereas in dynamic binding, components may *register* for events at run time and *deregister* for events when they are no longer interested. DYNAMO events are statically bound to up-calls of higher-level bottom services, thereby avoiding the dereferencing associated with function pointers.

There are also several different ways in which a component can announce an event. For example, there could be a central procedure that is used to announce all types of events. Alternatively, there could be a different procedure for each different type of event. Whereas the former is more flexible with respect to adding new events, the latter provides a greater degree of type checking and dependability. There is also a secondary efficiency benefit. The dispatching logic inherent with a central announcement procedure is avoided when there is a separate event manager procedure associated with each type of event. For these reasons, DYNAMO chooses multiple procedures for handling event announcements.

4. IMPLEMENTATION APPROACH

DYNAMO provides a highly-factored architecture for implementing a single component's computational-core class and its event manager. Figure 6 is a UML class diagram that provides a schematic view of the structure of a component. This schematic illustrates three different composition features:

1. static binding via template instantiation, indicated by the dashed dependency arrow;

2. virtual base classes, indicated by the solid inheritance triangle; and

3. abstract classes, denoted by the keyword interface and rendered in italics;

Static binding efficiently connects (1) the down-calls in one component to the top services in a subordinate component and (2) the event announcements in a component's computational-core class to the up-calls that are bound to these events by its event manager. By contrast, we use dynamic binding (through virtual inheritance) to connect the up-calls in an event manager to the bottom services of higher-level components. The abstract classes are used to organize the many parts in this design.

Each level has aspects associated with layering and implicit invocation. The schematic captures these aspects using three abstract classes: Component_i_top, Component_i_bot-

[2]The use of these terms should not be confused with their use to describe method binding mechanisms in object-oriented programming languages.

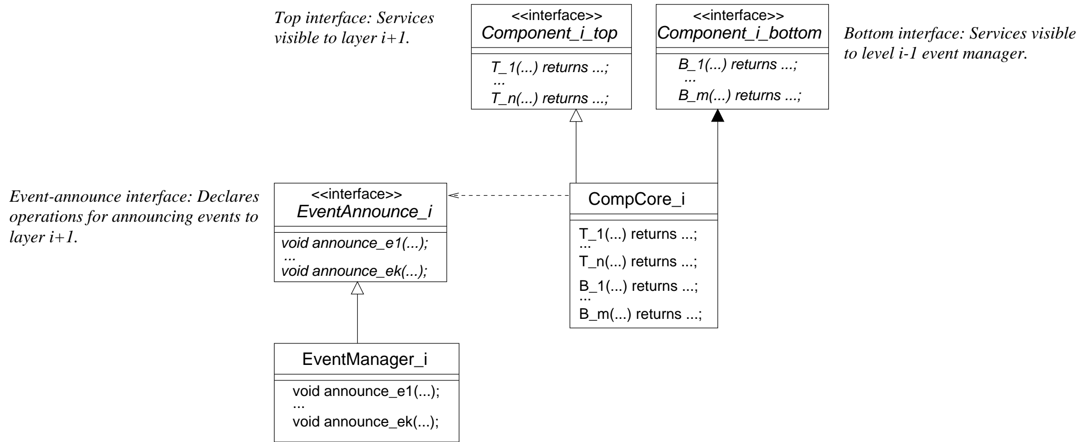

Figure 6: Object-oriented decomposition of a componenet in our architecture

tom, and EventAnnounce_i. These classes specify the interfaces of (respectively) the top and bottom services of component i in the assembly and the announcement procedures invoked for each distinct type of event. These abstract classes merely provide the signatures for the operations. Methods for these operations are provided by the two concrete classes—$\mathsf{CompCore_i}$ and $\mathsf{EventManager_i}$: $\mathsf{CompCore_i}$ provides the functionality for this component, as suggested in Figure 5; whereas EventManager_i contains the code that binds event requests to procedures.

Because the component interface is provided using several abstract classes, each may be refined (i.e., provided with methods) separately and then combined. For example, by providing an $\mathsf{EventManager_i}$ class whose announce methods invoke the bottom services of $\mathsf{CompCore_{i+1}}$, we can bind $\mathsf{CompCore_i}$ events to $\mathsf{CompCore_{i+1}}$ bottom services without having to modify any of the code for $\mathsf{CompCore_i}$ or $\mathsf{CompCore_{i+1}}$. When a designer constructs the code for $\mathsf{CompCore_i}$, she will write the code to announce events when information needs to be propagated up to higher-level components; however she will not need to know how these events are bound, as this knowledge is hidden in the event manager.

5. OPTIMIZATIONS

The DYNAMO implementation architecture allows designers the dependability benefits of the assembly architecture, with its support for layering and implicit invocation, without paying the efficiency price incurred by encapsulated layers and explicit event managers. Designs that conform to this architecture avoid two sources of inefficiency:

1. the binding of event announcements to up-calls, and
2. the binding of down-calls to lower-level top services.

Naive implementations of these two bindings results in an extra dereferencing to resolve the target of each invocation. We now show how both of these dereferencings can be avoided, and we show how to do so without complicating the binding of up-calls to bottom services in higher-level components. To aid in the exposition, we use an example (two-level) assembly, depicted in Figure 7. In the figure, the classes at the top of the diagram represent the level-2 component, and the classes at the bottom represent the level-1 component.

5.1 Optimizing away the event manager

First we demonstrate how to optimize away calls through the event manager. Having defined $\mathsf{CompCore_i}$ and $\mathsf{EventManager_i}$ separately, we can compose them using either parameterized inheritance or multiple inheritance, as discussed in Section 2.3. We chose to use parameterized inheritance in this case, because doing so allows us to optimize away the entire protocol of event announcement and dispatch, leaving explicit up-calls where initially there were calls to announce methods. To see why, consider the example assembly in Figure 7. In this example, the class EventManager_1 provides announce methods that make up-calls to bottom services of CompCore_2. We implement the dependency from $\mathsf{CompCore_1}$ to $\mathsf{EventManager_1}$ using parameterized inheritance as follows:

```
class EventManager_1 :  ...  { ...  }
template <class EVENT_MANAGER>
class CompCore_1 : public Component_1_top,
                   private EVENT_MANAGER {
 public:
  void T_1_l1(...)
   { ...  EVENT_MANAGER::announce_e1_l1(...); ...}
  ...
  void T_n_l1(...)
   { ...  EVENT_MANAGER::announce_ek_l1(...); ...}
};
```

Observe that CompCore_1 is defined as a mixin class, parameterized by a class that must conform to the $\mathsf{AnnounceEvent_1}$ interface. Now consider the snippet:

```
CompCore_1<EventManager_1> sys;
sys.T_1_l1();
```

Assuming that this code is compiled using an inlining compiler[3], the invocation of method announce_e1_l1 will be optimized away, leaving the body of this method at the call site. Because the body of an announce method invokes one or more up-calls, this trick effectively replaces the protocol of

[3]C++ supports the idea of inlined methods and functions. A method or function is inlined when its body is compiled in place at the site of its invocations, thereby avoiding the overhead of procedure calling.

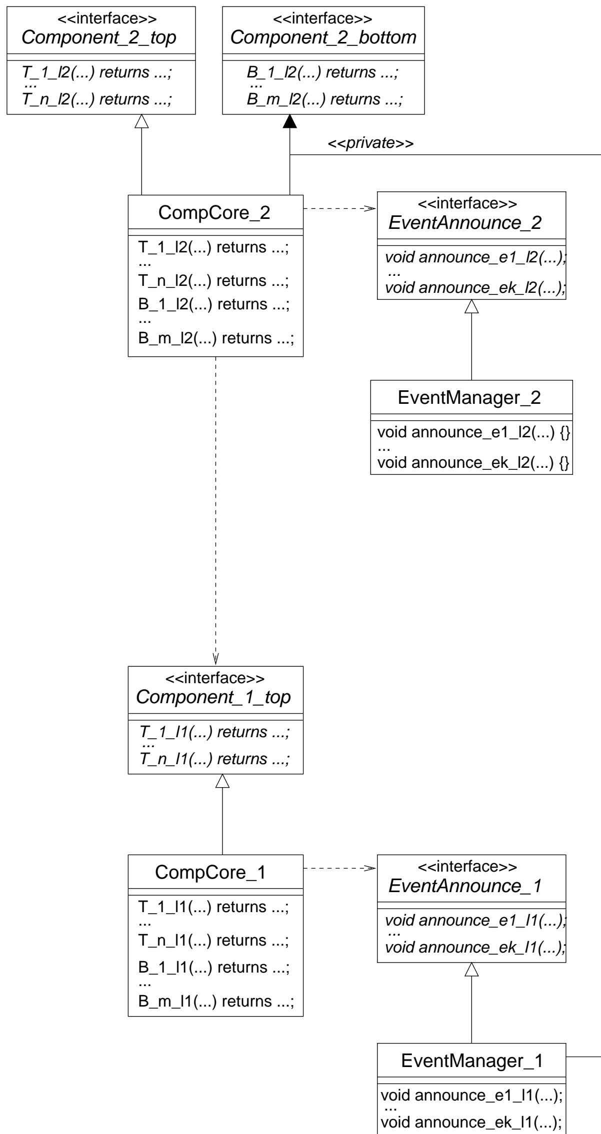

Figure 7: Framework for layered assembly

event-announcement and dispatch with the actual up-calls that result from that protocol.

5.2 Optimizing binding of down-calls to top services

The optimization in the previous section was concerned with entities in the same component. By contrast, optimizations concerned with the binding of down (and up) calls to top (and bottom) services are inherently inter-level. Using Figure 7 as a reference, we now describe how the indirection involved in binding down-calls to lower-level top services can be optimized away. The binding of up-calls to higher-level bottom services is treated in Section 5.3.

Consider the following elided C++ code[4] that defines two of the classes and template classes in Figure 7.

[4] We elided (using ellipses) details that are not necessary for understanding the binding of down-calls to top services. These details are provided in Section 5.3.

```
class EventManager_2 :  public EventAnnounce_2 {
 public:
  void announce_e1_l2(...)  {}
  ...
  void announce_ek_l2(...)  {}
};

template <class SUBORDINATE_COMPONENT,
          class EVENT_MANAGER>
class CompCore_2 : public Component_2_top,
                   public ...,
                   private SUBORDINATE_COMPONENT,
                   private EVENT_MANAGER {
 public:
  void T_1_l2(...)
   { ...  EVENT_MANAGER::announce_e1_l2(...); ...
     ...  SUBORDINATE_COMPONENT::t_1_l1(...)...
   }
  ...
  void B_m_l2(...)
   { ...  EVENT_MANAGER::announce_ek_l2(...); ...
     ...  SUBORDINATE_COMPONENT::t_3_l1(...); ...
   }
};
```

Because CompCore_2 invokes lower-level services by explicit invocation, it needs direct access to these services. This direct access is provided by parameterizing CompCore_2 by the lower-level component with which it will communicate. Whereas CompCore_1 required only one template parameter (for its event manager), CompCore_2 requires two template parameters: one for its event manager, and one for the component that provides lower-level services [5].

With these definitions in hand, we can specify the assembly of the two components as follows:

```
CompCore_2< CompCore_1< EventManager_1 >,
            EventManager_2 > sys;
sys.T_1_l2(...); // Example invocation
```

As we saw in the previous section, if we are using an inlining compiler, the down-call to operation `t_1_l1(...)` in method `T_1_l2` will be replaced with the code for method `t_1_l1(...)`. Observe also that EventManager_2 defines all of its announce methods as empty methods. Because we use parameterized inheritance to integrate an event manager into the computational-core class that uses it, the presence of these empty methods causes all of the invocations of the announce methods for component 2 to be optimized away.

5.3 Binding up-calls to bottom services

The preceding optimizations rely upon being able to resolve the binding of up-calls in an EventManager to bottom-services of a higher-level component without requiring the announce methods to explicitly name this component. For if it was necessary to name the higher-level component, then the lower-level component would need to be initialized with a pointer to this component, and for deep assemblies, this would require multiple explicit pointers. This subsection describes a mechanism for this binding under such constraints.

The binding of up-calls to bottom services is specified in the announce methods provided by a given EventManager.

[5] This is how GenVoca composition is traditionally implemented in C++ [2, 9].

To see how we achieve this binding without requiring the generated components to maintain pointers to each other, refer again to Figure 7. By privately inheriting the virtual base class Component_2_bottom, EventManager_1 plays the role of a client class in a diamond pattern of multiple inheritance, in which class Component_2_bottom is the root of the diamond. Similarly, CompCore_2 (once instantiated with its event manager) plays the role of the server class in the diamond pattern. For the diamond composition to work, CompCore_2 must also use virtual inheritance with respect to Component_2_bottom.

We can now fill in details that were elided in earlier examples. Consider first how the event manager is declared.

```
class EventManager_1 :  public EventAnnounce_1,
              private virtual Component_2_bottom {
 public:
  void announce_e1_l1(...)
   { ...  Component_2_bottom::B_m_l2(...); ...  }
  ...
  void announce_ek_l1(...)
   { ...  Component_2_bottom::B_3_l2(...); ...  }
};
```

Observe that EventManager_1 defines announce methods that invoke bottom operations in CompCore_2. This is possible because EventManager_1 inherits the bottom interface (Component_2_bottom of CompCore_2.

```
template <class SUBORDINATE_COMPONENT,
          class EVENT_MANAGER>
class CompCore_2 : public Component_1_top,
                   public virtual Component_2_bottom,
                   private SUBORDINATE_COMPONENT,
                   private EVENT_MANAGER {
 public:
  void T_1_l2(...)  {...}
  ...
  void B_m_l2(...)  {...}
};
```

With these definitions in place, we may now declare a variable (sys) that is an instance of the two-level assembly in Figure 7 using the following instantiation:

```
CompCore_2< CompCore_1< EventManager_1 >,
            EventManager_2 > sys;
```

Nowhere in this instantiation do we need to express the binding of up-calls to bottom services, as the binding is performed implicitly, according to the diamond property.

This use of the diamond property accomplishes the binding of up-calls in one component ($\mathsf{Component}_1$) to bottom services in a higher-level component ($\mathsf{Component}_2$). This use of multiple inheritance implies one dereferencing in order to dispatch the virtual function; however, no other dereferencing is required. Specifically, $\mathsf{Component}_1$ does not need to be initialized with (and does not need to manage) a pointer to $\mathsf{Component}_2$, as this management is performed automatically by the C++ compiler.

6. STATUS AND CONCLUSIONS

The DYNAMO project's primary concern is with the assembly of dependable and efficient systems from components. The work described in this paper is part of a larger effort which includes a graphical design environment, static analyzers of assembly properties, an integrated tools architecture, and a run-time monitoring infrastructure.

The graphical design environment consists of a UML editing tool called ArgoUML [1], originally developed at the University of California at Irvine. Using a UML Class Diagram, designers specify the services provided by each component and its externally visible state. Moreover, the Object Constraint Language (OCL) [11] can be used to declaratively indicate intercomponent constraints by annotating associations in the diagram. A constraint can be used to denote how one component depends on the state of its associate. Ultimately, state changes are realized as event announcements in DYNAMO's implicit invocation implementation. Designers can also specify intercomponent interaction protocols using ArgoUML's Statechart diagrams annotated with OCL.

The DYNAMO tools architecture supports the translation of UML design information into a common intermediate representation, which we devised, called Para [8]. Tools which wish to access the design information can then provide translators from Para into their own input formats. One such tool that has been integrated into the DYNAMO design environment is SMV, a model checker originally developed at Carnegie-Mellon University [5]. SMV can analyze designs and point out flaws, for example, detecting situations where intercomponent protocols can lead to unexpected behavior.

Another tool we are building converts Para into C++ using the implementation constructs described in this paper. That is, UML models are automatically converted into classes and templates and organized into layers using parameterized and virtual inheritance. There is no need for designers to directly confront these daunting constructs.

The DYNAMO run-time environment includes the ability to monitor system execution and report anomalies. Although some dependability problems can be determined statically with tools such as SMV, others require run-time detection. In DYNAMO, a *gauge* is a component visible to the end user or system administrator that reports on overall system status, including aspects such as resource consumption (memory use and bandwidth). Because status information is provided as a central part of component specification, no special treatment is required to support gauges.

Complexity is the enemy of dependability. If a system is so complex that it cannot be easily understood, then how can its behavior be guaranteed. Unfortunately, systems which are both simple and powerful are usually also inefficient. DYNAMO systems reduce complexity via the use of a layered, implicit-invocation architecture. Efficient implementation is provided by a stylized use of parameterized and virtual inheritance. The combination of these features is our candidate for meeting the dependability–efficiency tradeoff challenge.

Acknowledgements

This material is based upon work sponsored by the Defense Advanced Research Projects Agency (DARPA) and supported by the Air Force Research Laboratory under Contract No. F30602-00-2-0618. Any opinions, findings and conclusions or recommendations expressed in this material are those of the author(s) and do not necessarily reflect the views of DARPA or the United States Air Force. In addition, partial support for the second author was provided by

NSF grants CCR-9984726 and EIA-0000433.

We would also like to thank the Yamacraw Institute for supporting the DYNAMO project. We appreciate implementation support provided by Corinne McNeely, Terry Shikano, and Patrick Yaner.

7. REFERENCES

[1] ArgoUML. http://argouml.tigris.org.

[2] D. Batory and B. Geraci. Composition validation and subjectivity in GenVoca generators. *IEEE Trans. Softw. Eng.*, 23(2):67–82, Feb. 1997.

[3] D. Batory and S. O'Malley. The design and implementation of hierarchical software systems with reusable components. *ACM Trans. Softw. Eng. Meth.*, 1(4):355–398, October 1992.

[4] G. Booch, I. Jacobson, and J. Rumbaugh. *Unified Modeling Language User Guide*. Addison Wesley, 1999.

[5] K. L. McMillan. *Symbolic Model Checking: An Approach to the State Explosion Problem*. Kluwer, 1993.

[6] D. Parnas. Designing software for ease of extension and contraction. *IEEE Trans. Softw. Eng.*, 5(2), 1979.

[7] M. Shaw and D. Garlan. *Software Architecture: Perspectives on an Emerging Discipline*. Prentice-Hall, New Jersey, 1996.

[8] T. Shikano and S. Rugaber. Static analysis of yamacraw product-line assemblies. Technical report, Yamacraw Industrial Advisory Board Review, Apr. 2001. http://www.cc.gatech.edu/dynamo/papers/yamaapril.ps.

[9] Y. Smaragdakis and D. Batory. Implementing layered designs with mixin layers. In *Proc. of the 12th European Conference on Object-oriented Programming*, 1998.

[10] B. Stroustrup. *The Design and Evolution of C++*. Addison Wesley, 1994.

[11] J. Warmer and A. Kleppe. *The Object Constraint Language: Precise Modeling with UML*. Addison Wesley, 1999.