Modeling Real-Time Distributed Software Systems (Invited Paper)

Bran Selic
ObjecTime Limited
Kanata, Ontario, CANADA
bran@objectime.on.ca

Abstract

The "architecture" of a software system refers to its highest-level modular decomposition and the interrelationship patterns between its modules. An architecture serves as a blueprint for implementation and also as the chief determinant of a system's ability to evolve. Consequently, there is an increased interest in methods for specifying and validating software architectures. One such method is based on the ROOM modeling language which combines the object paradigm with modeling abstractions devised specifically for distributed real-time software. To overcome the treacherous "architectural decay" phenomenon, whereby, over time, software diverges increasingly from its specification, ROOM formally constrains the implementation to its architectural specification. This is achieved primarily through full automated code generation, a technique that is particularly challenging in real-time applications where stringent performance and memory requirements are the norm.

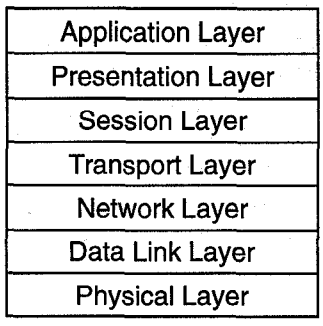
1: Introduction

For existing software systems, it is typically more cost effective to accommodate new requirements by incrementally modifying the software as opposed to replacing it completely. Over time, it is not unusual for the cumulative effort invested in such evolutionary development to exceed the initial development effort by several orders of magnitude. This creates a need to construct software in a way that facilitates incremental changes.

A common technique for achieving this is to decompose a system into a set of loosely coupled components each of which can change more or less independently of the others. Thus changes have only a localized effect and low cost. For instance, in communications equipment, it is generally wise to separate out the software that interacts directly with transmission hardware since that is a rapidly evolving technology.

Isolating elements that are likely to change is part of the more general problem of partitioning a system into multiple domains with different concerns. In addition to changeability, this partitioning may be driven by other considerations, such as ease of construction and understandability. The set of major partitions and the pattern of relationships between them is often referred to as the *architecture* of a system.

A notable example of a system architecture is the seven-layer OSI reference model for communication systems [2]:



An architecture acts as a framework that determines the form and placement of all other components of a system. This means that a change to the architecture usually has major repercussions; it implies corresponding changes in a large number of components that depend on that framework. Hence, an architecture is the chief determinant of a system's ability to accommodate evolutionary development.

We identify two primary aspects of an architecture. The *structural* aspect deals with the high-level decomposition of a system into components and their usage relationships. The *behavioral* aspect covers the dynamics of the system; i.e., the high-level interactions between architectural components required to achieve the desired functionality.

Since an architecture deals with abstract high-level forms, it is a *generic specification* that defines an envelope of possible concrete variations. The genericity inherent in

0-8186-7515-2/96 $5.00 © 1996 IEEE
Proceedings of the 4th WPDRTS

Proceedings of the 4th International Workshop on Parallel and Distributed Real-Time Systems (WPDRTS '96)
0-8186-7515-2/96 $10.00 © 1996 IEEE

architectures can be exploited in two different ways. First, it can be used to define product families. Each product in a family is a variant within the envelope defined by the architecture and, consequently, it automatically satisfies the high-level requirements that drove the architecture. This use of architectures is, in effect, a very high-level form of *design reuse*. Second, architectural genericity can be used to control system evolution ensuring that the initial high-level system requirements remain intact.

There is no universally accepted formalism for modeling the elements of a software architecture. Architectures are typically rendered using informal means such as block diagrams supplemented by descriptive prose. As a result, such specifications are often misunderstood, neglected, and even ignored. More often than not, they are relegated to the status of project folklore, treated by the implementation teams as the outdated and oversimplified products of ivory-tower naivete.

As a result, even during initial construction, there are no guarantees that the specified architecture will be the one actually implemented. This effect is exacerbated during maintenance which is often done at a very detailed level by the least experienced developers who, faced with massive amounts of code, have little awareness of higher-level concerns. The result is a gradual deterioration of the desired architecture which we refer to as *architectural decay.*

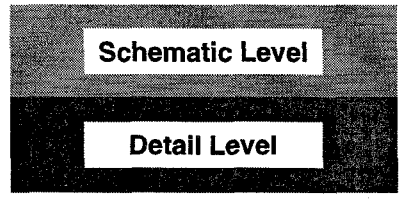
In this paper, we describe a practical approach for specifying the architectures of distributed real-time systems that overcomes this problem: architectures are explicit, formally specified, and enforceable. This approach uses an object-oriented, high-level specification language called ROOM (an acronym for Real-time Object-Oriented Modeling) which can be used to automatically derive implementations.

2: The ROOM Language

To avoid architectural decay, architectural specifications must remain current not only during initial analysis and design, but also throughout implementation, maintenance, and evolutionary development. The simplest way to guarantee this is for the high-level architectural specification to be an integral part of a complete implementation specification.

By definition, a complete specification encompasses all levels of a system, from the architectural down to the finest implementation detail. This, however, creates the problem of separating the "forest" (i.e., the architecture) from the "trees" (implementation detail). That is, the architecture is easily obscured by the overwhelming mass of individual programming language statements.

ROOM overcomes this problem by defining a system specification across two distinct but formally correlated levels. The upper level, called the *Schematic Level* (due to its graphical syntax), is used to specify the higher scopes of a system. The lower level, called the *Detail Level*, serves to define finer-grained implementation detail. The Detail Level specification is nested within, and constrained by, the Schematic Level specification.



To make the architectural specification more easily discernible, ROOM uses a graphical syntax at the Schematic Level (such as the OSI diagram shown earlier) supplemented by formal textual annotations for detail that is not easily rendered graphically. The Detail Level language, on the other hand, addresses detail that a typical implementation language covers (e.g., basic arithmetic). Rather than add yet another programming language to the current milieu, an existing language, such as C++ or Eiffel, can serve this purpose. The only requirement is that a prescribed set of "crossover" objects be defined in the Detail language to formally bridge the gap between the two levels.

This separation of language levels reduces the conceptual load on users. They are not faced with a single complex notation combining a forbidding number of different constructs whose differing degrees of significance are obscured by a common syntax. It also allows different semantic forms to be applied at different levels. For example, the semantics of inheritance are different for the two levels.

Another important characteristic of ROOM is that it is an object-oriented language. This enables the full spectrum of powerful features inherent in the object paradigm (such as encapsulation, polymorphism, and inheritance) to be used even at the Schematic Level (as opposed to being restricted just to the Detail Level).

In the remainder of this paper we will describe primarily the modeling concepts of the ROOM Schematic Level.

2.1: The Domain of ROOM

Different application domains require different architectural forms [3]. Rather than try and cover the full spectrum, ROOM focuses its Schematic Level language on just one domain: distributed real-time systems characterized by *event-driven* or, *reactive* behavior.

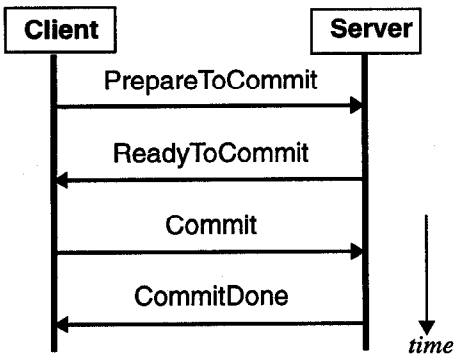
Proceedings of the 4th International Workshop on Parallel and Distributed Real-Time Systems (WPDRTS '96)
0-8186-7515-2/96 $10.00 © 1996 IEEE

A defining characteristic of such systems is that they are expected to respond in a timely fashion (the definition of "timeliness" depends on the application) to asynchronous events instigated by some external "real-world" event. Since the external world is often unpredictable, events can occur in any order and at any instant.

2.2: The Communication Model—Protocols

In ROOM, the occurrence of an event is signalled by the arrival of a message at an object. Message passing is the primary mode of communication between objects at the Schematic Level. It is convenient both for modeling the asynchronous nature of events as well as for dealing with distributed environments where shared memory may not exist. Note that message-based communication does not necessarily imply that all communication is asynchronous; both synchronous communication (i.e., like the Ada rendezvous) and asynchronous communication are supported in ROOM.

It is common for a collaboration between two objects to take on a specific pattern or *protocol*. A protocol comprises an ordered sequence of invocations and replies between two objects (e.g., a two-phase commit protocol).



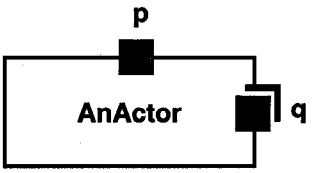
Each message that is exchanged in the course of a protocol sequence consists of a named *signal* and an optional passive data object that specifies one or more parameter values.

It is standard for the same protocol specification to be used by many different objects. For example, different clients of a shared server will require the same protocol to interact with the server. Rather than recreate the same specification each time, it is more practical to define a single reusable specification and then reference it as necessary. In the object paradigm, this is provided through the *class* concept. A *protocol class* in ROOM defines a reusable message protocol specification. It consists of a list of message types (<signal, data class> pairs), their individual directions (incoming or outgoing), and an optional specification that defines all valid message exchange sequences.

Specifying protocols as classes suggests that protocols can be related to each other through inheritance creating a protocol class hierarchy. This provides the ability to define abstract protocols that can be specialized in different ways through subclassing. This feature is extremely useful in early architectural work where the full details of an object collaboration are either irrelevant or unknown.

2.3: Modeling Structure

The primary structural element of ROOM is called an *actor*. An actor is a *concurrent* object responsible for performing some specific function. It is concurrent in the sense that it can operate in parallel with other actors.



An actor communicates with other objects through one or more interface objects called *ports*. Each port represents an instance of one protocol class. A single actor can simultaneously handle multiple different protocols. The set of interfaces on the outside of an actor define its *type*.

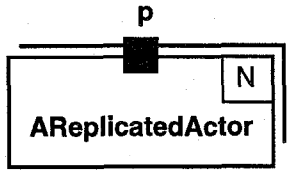
Types were originally introduced into computer languages as a way of protecting programmers from accidentally falling into the proverbial "mixing apples and oranges" trap. In principle, the type of an object characterizes its externally observable semantics so that its collaborators know what to expect.

Ports act as intermediaries between an actor's implementation, contained within the encapsulation shell, and its environment. The implementation only interacts directly with ports; it is decoupled from the environment so that the same actor specification can be used in a variety of contexts. Once again, the possibility of a reusable specification comes to mind. Thus, an actor is defined as an instance of an *actor class*. (Note that the class of an actor, which includes the specification of its implementation as well as its interface, is distinct from its type.)

A frequent situation occurs in practice when there are multiple instances of the same type of entity that need to be handled in some uniform fashion. In traditional programming languages, the array construct is used in those situations. ROOM handles this through *structural replication*. For example port **q** in the diagram above is a replicated port (indicated by the extra flange on the port icon). Actor

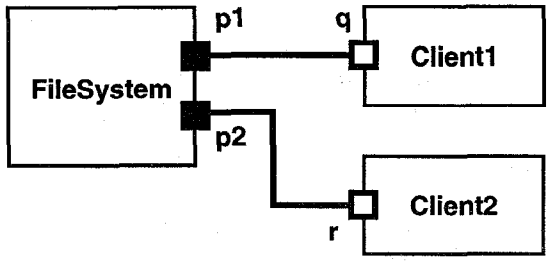
Proceedings of the 4th International Workshop on Parallel and Distributed Real-Time Systems (WPDRTS '96)
0-8186-7515-2/96 $10.00 © 1996 IEEE

instances can also be replicated if they are all instances of the same class



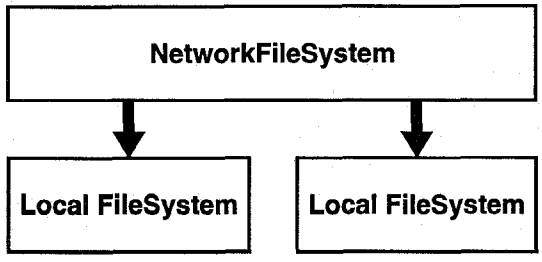
To make useful systems, it is necessary to combine one or more objects of different kinds into more complex functional aggregates. In ROOM, this is achieved through three primary composition mechanisms: bindings, layer connections, and containment. Bindings and layer connections are used to model communication relationships while containment captures compositionality relationships between actors.

A *binding* models a communication channel that connects two ports. Its graphical representation is a continuous undirected line that connects two ports. (The white fill on the two client ports indicates that these ports use an "inverse" of the protocol class at the other end of the binding.)



Only ports that have compatible protocols can be mutually bound.

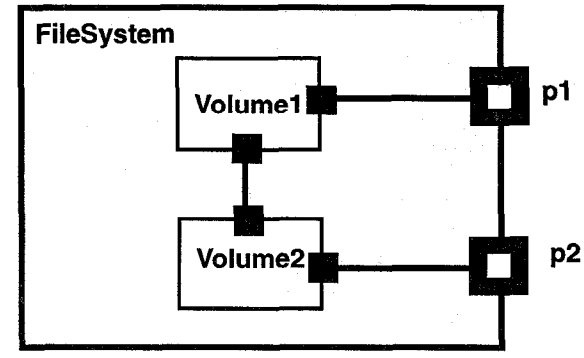
In contrast to bindings, *layer connections* are directed relationships and model situations in which there is an asymmetric dependency between two actors. This asymmetry is typically in the form of a client-server relationship: the client cannot function unless a server is present whereas the server can exist and function independently of any particular client. The graphical representation of a layer connection is a directed arc between two actors.



A layer connection connects one or more *service access points* (SAP) on the client to a *service provision point* (SPP) on the server. SAPs and SPPs are also instances of protocol classes like ports. Hence, all the rules of port connections apply. However, since the number of such connections tends to be very large, to reduce visual clutter, each layer connection typically represents multiple individual point-to-point connections. For the same reason, SAPs and SPPs are not rendered explicitly.

The pattern of layer connections and bindings that interconnect actors defines the complete set of possible communication relationships between actors. This is a crucial aspect of a software architectures since it explicitly identifies all the potential causality links between actors.

Architecture is a relative concept; a significant component in a large system can have an architecture of its own. From this it follows that it should be possible for actors to be decomposed into aggregates of more elementary actors. For example, the **FileSystem** actor from our earlier example might be decomposed as follows



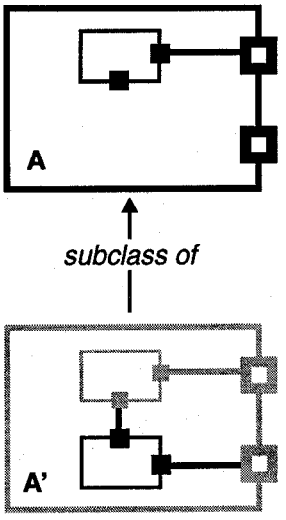
The component actors, **Volume1** and **Volume2**, might have their own internal structure consisting of other actors and so on. ROOM does not impose any limits on the number of levels of decomposition. This enables the specification of arbitrarily complex architectures. Note that, because of the impermeability of the actor encapsulation shell, the two clients of **FileSystem** (**Client1** and **Client2**), are completely unaware of its internal composition; that is, they cannot "see" its component actors and bindings.

Note that the two ports, **p1**and **p2**, that constitute the interface of **FileSystem**, are just relay points that simply redirect incoming messages to the appropriate component actors. Accordingly, such ports are called *relay ports*.

An actor class can be subclassed. Subclasses automatically inherit all the attributes of their superclasses (including its implementation) and may add new attributes or modify existing ones. An example of the structures of an actor class (**A**) and one of its subclasses (**A'**) is shown in below. In the subclass, all inherited attributes are drawn

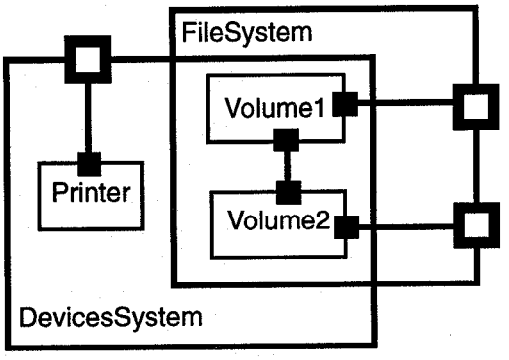
Proceedings of the 4th International Workshop on Parallel and Distributed Real-Time Systems (WPDRTS '96)
0-8186-7515-2/96 $10.00 © 1996 IEEE

with a lighter pen while new attributes are rendered in black.



The decomposition strategy that we have defined so far, leads us to a pure tree-like decomposition. While this has an appealing simplicity, it is generally inadequate to describe most complex systems. The reason for this can be traced to the nature of composite objects such as our **FileSystem** example. In essence, the purpose of this object is to define a convenient abstraction, or "view", of the functionality that is provided by the hidden components inside. In most complex systems, it is difficult to define just one way of decomposing an architecture into abstractions. This is because there are usually many different and equally valid views of the same system. For example, the two volume components of the **FileSystem** actor may simultaneously be part of some abstract "devices control" system that manages them and other devices.

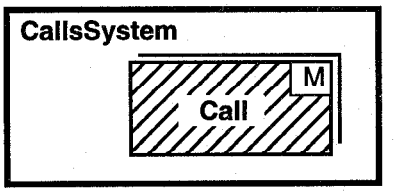
This means that, if desired, the transitive decomposition structure of a ROOM actor can be described by a directed acyclic graph rather than just a simple tree. This feature is called *multiple containment* since some actors may simultaneously be in more than one container.



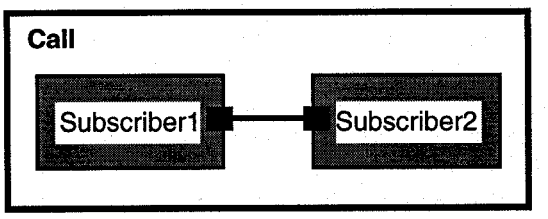
All the structures discussed to this point have been static or "hard-wired". However, many systems (especially real-time systems) are embedded in a dynamic environment that may require the dynamic establishment of transient relationships between two or more objects.

An example of such a relationship is the abstraction that we refer to as a "telephone call". This is simply a temporary relationship that is established between two subscriber lines. The object paradigm allows us to cast such abstract intensional entities in the form of concrete software objects. If we choose to model calls as objects, then these objects should be created and destroyed as needed since telephone calls are usually transient objects. (This conserves resources—it is not feasible to dedicate separate resources to every possible call that can be made.)

For this purpose ROOM provides the concept of *optional actors* illustrated by the hashed replicated **Call** component below. In contrast to other components, optional actors are not created automatically with their container actor. Instead, they are created subsequently as appropriate, by the container, and destroyed when they are no longer needed. In the example below, the **CallsSystem** can contain anywhere between zero and a maximum of M **Call** actors.



Assume that a simple telephone call is represented by a composite actor that contains the two subscriber actors involved in the call.



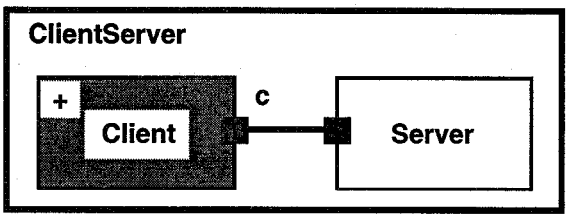
When defining this actor class, we cannot know in advance which of many subscribers will participate in a particular instance of a call. Therefore, the structure of the call actor must be generic, representing all possible calls. This means that the two components cannot be concrete actor instances, but are, in fact, actor "slots" into which specific instances will be placed at some point. Such placeholder components are called *imported actors* and they are indicated graphically by a darker shading.

Note that actor importation assumes multiple containment since the imported actors must exist elsewhere in order to be imported into a container.

Proceedings of the 4th International Workshop on Parallel and Distributed Real-Time Systems (WPDRTS '96)
0-8186-7515-2/96 $10.00 © 1996 IEEE

To facilitate the ability to capture product families, it is useful to provide an explicit mechanism for controlling genericity. In ROOM, this is achieved through *substitutable actors.*

The component actors that appear in a composite actor are normally instances of specific classes. To make such structures generic, component actors can be specified by type rather than by their class. A substitutable actor is identified graphically by a '+' icon.



For example, the actor dynamically imported into the **Client** slot above can be an instance of any actor class whose protocol corresponds to the protocol of port **c**.

2.4: Modeling Behavior

The point where the structural and behavioral dimensions meet is at the interfaces of an actor (ports, SAPs, and SPPs). With the exception of relay ports, which mindlessly shuttle messages back and forth, an interface can be directly manipulated by the *behavior component* of an actor. A behavior component may be attached to any actor, including a composite. To reduce clutter, it is not rendered explicitly in a structure diagram but is simply assumed to be directly attached to the (non-relay) interfaces of an actor.

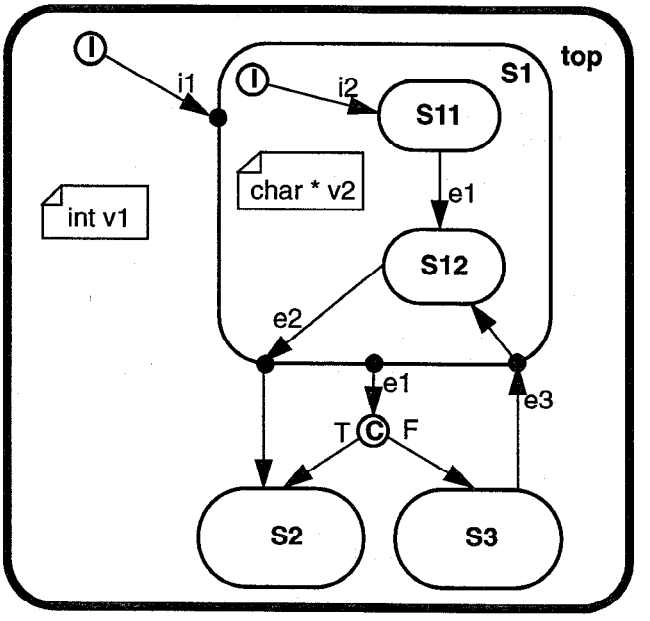
Except for external messages, the behaviors of actors are the ultimate source and sink of all messages. To send a message the behavior simply invokes a "send" function on the appropriate interface. The message then travels through bindings and, possibly, relay ports until it reaches an interface that is attached to the behavior of the actor at the other end. When a message is received by a behavior, it results in an invocation of a Detail Level function (encapsulated in a transition in a state transition diagram). The details of this function are specified in the Detail Level language of choice (e.g., C++). Thus, behavior interfaces also serve as the "crossover" points between the Schematic Level and the Detail Level mentioned earlier. At the Schematic Level, these interfaces appear as anchor points for structural relationships; at the Detail Level, they take on the form of passive data objects with the capability to send and receive messages.

Reactive systems require modeling formalisms that are quite specific. For this purpose, at the Schematic Level, ROOM uses a variant of hierachical finite state machines called *ROOMcharts*. A state-machine based formalism was chosen because state machines are relatively well known among practitioners and have been widely used to capture reactive behavior.

In ROOMcharts, an event is represented by the arrival of a message at an actor interface. Depending on the current state of the behavior, the type of signal in the message, and the interface on which it arrives, the event will trigger a transition that leads to a change of state of the behavior.

ROOM uses a *run-to-completion* model of event processing. This simply means that messages are processed one at a time; once message handling is commenced and until it is completed, no other messages are processed by that behavior[1] (this implies queueing of messages at the interfaces). The advantage of this approach is that it provides automatic mutual exclusion of event handlers and, thereby, significantly simplifies the behavior specification. It is justified in situations where the handling of events is relatively short—a property shared by the majority of reactive components.

Hierarchy in ROOMcharts means that a state can contain a state machine (e.g., state **S1** below).



As with structure, hierarchy allows us to address a complex problem gradually, one level of abstraction at a time.

In addition to hierarchy, ROOMcharts use several graphical conventions that can significantly simplify the graphical representation of complex behavior and, consequently, lead to more understandable and more reliable specifications. These are derived from similar features in the "statechart" formalism [4]. For example, transition **e1**

1. However, other actors may process their messages during that time.

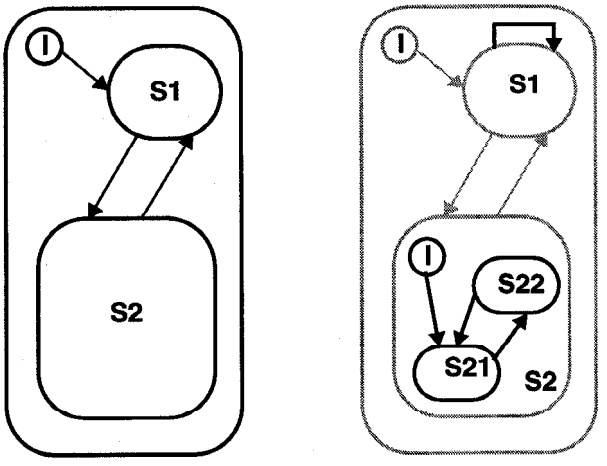
Proceedings of the 4th International Workshop on Parallel and Distributed Real-Time Systems (WPDRTS '96)
0-8186-7515-2/96 $10.00 © 1996 IEEE

in the above ROOMchart is a short-hand notation for two separate transitions, one emanating from state **S11** and the other from **S12**, triggered by the same event. This type of transition is known as a *group transition*.

Each transition can have an associated Detail Level program sequence that specifies what is to be done as part of the transition. This is where the contents of the incoming message can be processed and messages can be sent to other actors. As an option, it is possible to associate with each state an *entry action* (executed whenever that state is entered, regardless of which incoming transition was taken) or an *exit action*.

Extended state variables are instances of Detail Level passive data objects which are used by the finite-state machine to maintain auxiliary status information (e.g., the integer variable **v1** above). These objects can be accessed by the code inside transitions and entry and exit actions. Since actors are fully encapsulated, extended state variables are not accessible by other actors. Block-structured lexical scoping rules are used to control access to extended state variables within the state machine.

Another feature associated with ROOMcharts is inheritance.



Parent class **Subclass**

In this case, a subclass automatically inherits all the behavior attributes of the parent class and can refine that behavior by adding further attributes. The most common method of refining behavior is to extend an elementary state of the superclass into a full-fledged state machine.

Both additions and overrides are allowed for pragmatic reasons. This was based on the conclusion that for active objects it is generally not possible to guarantee behavioral equivalence of the parent class and its subclasses even if overriding is disallowed. For example, adding just one extra transition to a subclass (e.g., triggered by a timing signal), can break behavioral equivalence between a class and its subclasses.

3: Automatic Code Generation

If the objective of eliminating the architectural decay phenomenon is to be met, it must be possible to automatically derive a complete and *efficient* implementation of a system from its ROOM specification. This avoids the various pitfalls that occur when a design specification is translated into a programming language implementation.

The ROOM modeling concepts described above were specifically chosen with this objective in mind. In some cases it meant a trade-off between modeling power and efficiency. For example, ROOMcharts do not allow the specification of concurrent states, something which is possible in statecharts [4]. This is because the software realization of concurrent states often requires expensive and inefficient mechanisms (including open-ended broadcasting and synchronous execution).

The specific techniques used for code generation are beyond the scope of this paper but are described in [5]. In this section, we focus on the most significant properties of automatic code generation and describe some empirical results.

Code generation from ROOM models is implemented in the ObjecTime toolset and has been used successfully in a variety of embedded system applications and products.

The performance of the generated implementation depends on the application and its processing requirements. What we describe here, however, is the overhead that is inherent in the current code generation techniques.

The current generated code can process as many as 3,000 events per second per SPECint[1]. This means that a 20 SPECint machine could process up to 60,000 events per second. This works out to approximately less than 17 microseconds per event cycle. An "event cycle" includes the sending of a message from one actor to another, scheduling, and the dispatching of the receiver actor for execution.

In addition, up to 300 actor creation/destruction operations per second per SPECint can be handled. For our typical 20 SPECint machine, this amounts to as many as 6000 actor creations/destructions per second (170 microseconds).

The overhead for program store varies greatly with the target machine with RISC-based processors requiring more storage than CISC processors. The current average program store requirements for the virtual machine itself are approximately 100 Kilobytes on a RISC-type processor and 60 Kilobytes on a CISC processor. These sizes can be reduced significantly (e.g., 25 Kilobytes) by careful selec-

1. To factor out the diversity of target environments as much as possible, performance numbers are expressed in terms of rates per nominal SPECint.

Proceedings of the 4th International Workshop on Parallel and Distributed Real-Time Systems (WPDRTS '96)
0-8186-7515-2/96 $10.00 © 1996 IEEE

tion of default code libraries that are to be included with the generated code.

The overhead of an actor depends on the complexity of its internal structure. A typical actor instance, such as the one described in the diagrams above, will take up about 500 bytes.

A number of large applications have been developed using this approach, many of them done in the context of large legacy systems.

One typical project involved a team of over a dozen developers all collaborating on the same ROOM model for a telecom application. It was designed to interwork with other software running in the same target environment. The model made much use of industry-standard and proprietary C++ libraries.

The high-level model comprised approximately 80 actor classes, 100 protocol classes, and 400 data classes (in addition to external C++ class libraries). The detail spec consisted of 1,500 user-defined C++ code segments (specifying the details of event handling) containing over 21,000 non-commented C++ statements.

The code generation, generated a set of C++ modules consisting of a total of 144,000 non-commented lines of C++. This means that approximately 85% of the code of this system was automatically generated from the Schematic Level model.

On a single-processor SPARCstation 20 platform from Sun Microsystems, the ROOM model described above takes approximately 30 minutes to recompile from scratch. This includes all of the automatic code generation (about 10% of the overall time budget) as well as all C++ compilation. This time can be cut down significantly with the use of parallel compilation (an option in the high-level compilation system).

An incremental change that has only local implications (i.e., does not require the recompilation of other modules), can be turned around in as little 35 seconds. This assumes that all changes are done using the front-end design tool and not directly on the auto-generated code.

Both of the above compilation times are close to those that would have been experienced had straight C++ code development been used. This means that, once a code generation system is in place, the benefits of complete transformations can be obtained at a relatively low cost.

4: Related Work

Although there is a number of excellent architectural definition languages defined (e.g., [3]), the one that is closest in spirit to ROOM is the Regis/Darwin system developed at the Imperial College in London [6]. The other efforts are generally not targeted at real-time applications (which have stringent demands on time and memory) nor are they oriented towards code generation (hence, they are susceptible to architectural decay). There are, however, several significant differences between Regis and ROOM. Most notable, perhaps, is that ROOM has incorporated the object paradigm and can take advantage of its features at the highest levels. For example, entire architectures can be subclassed resulting in a very effective form of reuse at the highest levels of abstraction. In addition, ROOM provides a powerful behavioral modeling mechanism for dealing with reactive systems. Furthermore, ROOM's ability to model dynamic structures using multiple containment allows for a dynamic hierarchy as opposed to a purely static tree-structured one. On the other hand, Regis is a more open system that flexibly supports heterogeneous objects and environments in its specifications. We are currently working towards establishing a collaboration with the Imperial College group to incorporate the best of the two research thrusts.

5: Summary

We have described a practical approach to architectural modeling for distributed real-time systems. The approach is based on an object-oriented modeling language, ROOM, that permits automatic generation of implementations from specifications that incorporate complete and highly visible architectural specifications. The net result is an approach to real-time design that is both highly reliable and highly productive.

References

[1] Selic, B., Gullekson, G., and Ward, P., *Real-Time Object-Oriented Modeling*, John Wiley & Sons, New York, NY 1994.

[2] H. Zimmerman, "OSI Reference Model — The ISO Model of Architecture for Open System Interconnection," IEEE Trans. on Communications, vol. COM-28, no. 24, April 1980.

[3] Kogut, P. and Clements, P., "Features of Architectural Representation Languages," CMU/SEI-94-TR, Software Engineering Institute, CMU, Dec. 1994.

[4] Harel, D., "Statecharts: a visual formalism for complex systems," Science of Computer Programing, vol. 8, 1987.

[5] Selic, B., "High Performance Implementations from ROOM Models," Proc. 7th Annual Embedded Systems Conference, San Jose, CA., 1995.

[6] Magee, J. et al., "Specifying Distributed Software Architectures," Proc. 5th European Software Engineering Conference, Barcelona, Sept. 1995.

Proceedings of the 4th International Workshop on Parallel and Distributed Real-Time Systems (WPDRTS '96)
0-8186-7515-2/96 $10.00 © 1996 IEEE