6. LANGUAGES and PROGRAMMING of DISTRIBUTED REAL-TIME SYSTEMS

Real-time development has required that the programmers analyze software prior to execution to make sure that all execution time needs of software will be available when required. Resources that must be proven available include memory and CPU time. Traditional techniques for real-time development are very costly for a number of reasons [Nil95a[, [Nil96]:

- Greater care is required during development to perform the analysis required to guarantee compliance with real-time constraints.
- Because most general purpose programming environments do not support real-time performance, developers of real-time applications are forced to use special operating systems and development environments. These environments lack of robust and powerful development tools that are available to developers of more traditional systems.
- Since the techniques for analysis of real-time performance are intrinsically machine dependent, extra analysis effort is required to maintain real-time software in the presence of continual processor upgrades and integration of I/O devices.
- Because it is not practical accurately predict worst-case execution times, the real-time developers must reserve resources based on conservative upper bounds rather than actual worst-case requirements.
- Even it is possible to accurately determine worst-case execution times, execution time must be reserved for every task's worst-case performance. Note that for most tasks, typical execution time are much smaller than worst-case execution times.
- Because of difficulty of proving availability of dynamic memory, many developers of hard-real-time systems avoid all use of dynamic memory management. This requires that all of the memory required to satisfy each task's worst-case needs must be permanently allocated prior to execution. Memory not currently in use sits idle rather than contributing to the system's functionality and capacity.
- The scheduling techniques that are typically used in real-time systems are unable to promise 100% CPU utilization. Furthermore, context switching is much more frequent in many real-time systems than is typical of traditional systems. As a result, context switching and scheduling overhead represent a significant fraction of real-time system's total workload.

For all the costs and difficulties associated with the development of real-time applications using these traditional methodologies, you would think that the resulting benefits are well worth the effort. But in fact, most real-time systems fall sort of the ideal in terms of functionality and flexibility. Real-time software is notoriously brittle and feature poor. **There has to be a better way**.

Object distribution has been regarded as an essential technology for distributed computing. But few work has been done on distributing real-time objects. In this chapter we summarize the main results have been achieved in this field.

The first section of this chapter **introduces** the problems with large complex distributed real-time systems. The second part of the chapter summarizes several existing techniques for **representing** system behavior and describes the key features of the advanced approaches for capturing behavior for a large, complex, real-time, computer-based system within a multi-domain system design capture and analysis methodology. Methods commonly used to represent the system behavior at various levels of abstraction and/or different phases in the design process are briefly described. Following the summary of existing techniques we concentrate on the **modeling and programming** aspects of real-

time distributive systems. We introduce the main concepts of the real-time ANSA programming model, the ROOM language and modeling and finally the JAVA real-time extension.

6.1. Introduction

Large-Scale Systems. The complexity and sheer magnitude of modern computer systems including large scale advanced military, telecommunications, aerospace, and financial applications require that a disciplined structure approach be employed for their development. The principal objective of the system development process is to establish a design which satisfies the system requirements and constraints while optimizing the key tradeoffs and issues associated with system functionality, behavior, and implementation.

Real-Time Distributed Systems. It is the stringent timeliness and performance requirements of real-time applications that are the primary source of problems posed by real-time applications. The services provided by existing distributed system environments predate the present concerns of real-time applications and provide insufficient and in-appropriate services for supporting them. For example, current standards for distributed processing, such as the OSF DCE, OMG CORBA and ISO RM-ODP make no mention of real-time issues.

Many of the more ambitious "industrial-strength" sorts of applications have associated real-time constraints. These applications include video conferencing integrated with distributed white boards, virtual reality, voice processing, full-motion video and real-time audio for instruction and entertainment, and distributed video games. More importantly, the next generation Web client will have even more real-time requirements. Future set-top devices will connect home televisions to the Web by way of cable TV networks. Besides all of the capabilities just mentioned, these systems will also support fully interactive television applications.

It is important to recognize that the embedded computing market is quite large. Industry observers have predicted that by the year 2010, there will be ten times as many software programmers writing embedded systems applications as there will be working on software for general purpose computers [Ran94].

System Behavior. Large-scale real-time distributed systems can be complex to describe construct, manage, understand, and maintain. Current research approaches to reducing this complexity separate software structural and behavioral descriptions. It is, therefore, important to identify and formally describe software behavioral views and structural patterns; these patterns and/or views, then, can be used in software systems analysis, specification, design, and configuration. Much research has been done on software structures and their patterns, characterizations, and classifications. Little or no attention, however, has been given to the identification and analysis of software behavioral views that can be used as a basis for software behavioral requirements engineering.

Problems. Distributed real-time processing places a set of unique requirements on a system environment including predictability, user control, timelines, mission orientation and performance [Li94].

Predictability. Predictability is the tendency of a system to perform a set of operations in a well-defined, or deterministic fashion, so that each of these operations' timing requirements are satisfied. A fully predictable system performs operations in the same amount of time, every time, independent of surrounding conditions. Conversely, a fully nondeterministic system is one in which operation times have non guaranteed upper bound. Predictability applies to every level of the components of a real-time distributed system environment. Such an environment must provide a certain degree of predictability, even though it is not always possible to be fully predictable, to support any useful real-time performance guarantee.

User Control. User Control means a user has ultimate control of the behavior of a system. This feature comes from the fact that many real-time applications are embedded system (which are often

static systems, and therefore it is possible to control the systems' behavior) and that real-time applications have immense behavior diversity (therefore it is impossible to use one fixed system behavior for many real-time applications). The simples method of user control on system behavior is probably the choice of priorities for real-time tasks. By allowing a user to indicate the relative priorities of tasks, the user can affect throughput and/or responsiveness goals for the system on a much finer granularity than by a best effort approach. Users may also be allowed to select the scheduling policy, preallocation of system and application resources to critical services and so on.

Timeliness. Real-time applications are different from the no-real-time paradigm of computation in that they impose strict requirements on the timing behavior of the system. The correctness of a real-time system depends not only on the functional behavior of the system, but also on the temporal behavior as well. A real-time system environment must provide mechanisms which take these time related issues into account and must help application program to meet these time constraints. A simple example is to allow an application to associate deadlines with real-time activities, and the system employs a deadline based scheduling policy to help the deadlines to be met or to identify and cancel obsolete operations. Other required functions include the description and enforcement of temporal relations among related computation activities.

Mission Orientation. Mission Orientation means that an entire distributed computer system is dedicated towards accomplishing a specific purpose through the cooperative execution of one or more application programs distributed across its nodes. In the real-time sense, mission orientation also means mission critical - the degree of mission success is strongly correlated with the extend to which the overall system can achieve the maximum dependability regarding real-time constraints. In its simplest form, mission orientation requires that a priority or deadline associated with a mission has global meaning when it spans the network. More generally, global importance and urgency characteristics are propagated through the system, for use in resolving contention over system resources according to application defined policies.

Performance. Real-time applications have stringent raw performance requirements. The optimized integration of application software and its supporting environment is desirable. This is in contrast with the popular layered design for non-real-time applications. Also, real-time applications often require trading off modularity, flexibility and functionality to maximize performance.

6.2. Representation

Following is a summary of the methods commonly used to represent the system behavior at various levels of abstraction and/or different phases in the design process. This summary (based on [Kar94]) does not address the evaluation capability of the methods (most system behavior representation methods are intended as analysis techniques); instead it focuses on the ability to represent behavior information throughout the system development life cycle such that analysts can use it to perform the evaluation. The description of the methods is at a summary level with additional detailed information referred to in appropriate references. Information on how these methods are employed and their automation support is also included.

Real-Time Structured Analysis. Real-Time Structured Analysis (RTSA) [Hat88] is an extension of the traditional Structured Analysis (SA) method [Dem78]. By adding a control model (to support the specification of the functions' behavior) to a process model (the traditional SA concept), the method provides a mechanism to express system behavior. The process model includes Data Flow Diagrams, which use processes to represent the transformation of information; data flow to represent the flow of information between processes; and data stores to represent collections to frequently-needed information. The control model includes Control Flow Diagrams, which are paired with DFDs and contain control flows, control stores, and the same processes used in the DFDs. While typically used for system software specification, RTSA's ability to model both data and control flow, coupled with its hierarchical representation of processes, make it an efficient way to model large complex systems.

Petri Nets. Petri nets, which are a formal means of representing the state of large complex systems, can be used to represent behavior of which states and actions are shown as places and transitions, while the arcs that connect them show the flow of tokens (symbolizing control) from one mode to the next. A net's marking, or map of token locations at a given instant, shows the state of the system's processes. Each process has preconditions and postconditions; when a node has the required number of tokens at its precondition, it is said to be enabled, meaning that it can move tokens from its input (preconditions) to its output (postconditions). Nodes can be nested in a hierarchy, which simplifies analysis of large systems. Additions include semantic meaning to the nodes and arcs through the use of predicates and colored nets [Rei82], [Hub86], [Lin90], [Chi90]. These additions allow the modeling of different kinds of flow, for example control or data flows. In addition, colored tokens are unique and differentiable entities. This increases the power of transition preconditions, since a transition requirement can include a specific "color" of token to satisfy its preconditions.

If one compares a statechart [Har87] and its corresponding Petri net, one clearly sees the correspondence between concepts in the two models. However, since Petri nets introduce more detail, there are several possible Petri nets that correspond to a given statechart. In particular, Petri nets make explicit the implicit parallelism of statecharts; the designer must then select a corresponding Petri net that has the desired degree of parallelism.

Hierarchical Graphic System Design Capture And Simulation Method. HGSDCSM allows the system's functions and behaviors to be specified in an executable format [Cas90], [Cas93]. HGSDCSM draws on both RTSA (for modeling of functions and control) and Petri nets (to model primitive control structures). By integrating features of these other methods, HGSDCSM can address large, complex real-time systems where design options must be analyzed and weighted as early as possible and at all levels of detail. In order to handle the size and complexity of such systems, this method emphasizes both the partition of design information into domains and the hierarchical representation of such information. The design domains include the Functions/Data Flow View, which describes the system's functions and associated data stores and paths; the Control Flow View, which describes the order in which functions operate, including any dependencies among the functions; and the Resource View, which describes the system's hardware, software, and human resources. When mapped together, these views describe the entire system and provide a means of analyzing behavior through simulation.

System Requirements Engineering Methodology. SYSREM supports the specification of system requirements and the allocation of these requirements to the data processing subsystem [Alf85]. Using Behavior Diagram notation, SYSREM represents the system behavior as time functions which accept input arriving during a time interval and transform it into output; for the analysis of different time scenarios, multiple behavior models must be built. Completion criteria for and performance of the functions are also specified. SYSREM's Behavior Diagram offers the capability to represent the behavior of system functions in response to conditions and events. Its notation specifically addresses system development issues such as scaleability, complexity, clarity, completeness, consistency, and executability. In this context, the system's behavior is the result of the response process to external stimuli. Thus, SYSREM provides a means to specify system requirements such that a candidate can be simulated. The hierarchical structure of the Behavior Model allows the system behavior to be observed at many levels of detail, which supports the analysis of large complex system designs.

Z Formal Specification Language. Z notation is a formal specification language to specify information systems and reason about their behavior without constraint due to the syntax or structure of a particular implementation. As a mathematical language, Z notation mediates between software requirements, normally written in natural language, and the code that fulfills these requirements [Spi92]. Thus, Z is used primarily in software design. In Z, the system specification is represented by a collection of schema, each used to specify static or dynamic aspects of the system. Static aspects include the states the system can occupy or the variant relationships that are maintained as the system moves from one state to another, while dynamic aspects include the system's operations, the

relationship between input and output variables of the operations, and the change of state (from before to after execution of the operation).

Communicating Sequential Processes. CSPs model a system as a process which interacts with its environment by means of atomic events. The CSP approach is primarily used for software design, although it could be used to represent the behavior of large complex systems. CSP compresses an algebra of processes, together with a related set of semantic models for capturing different aspects of behavior. Patterns of behavior are described by pre-defined events, or atomic actions without duration [Hoa85]. Time-consuming behaviors are represented by a start/finish event pair. The summation of all system processes and their interactions represents the complete system, which is itself considered a process representing and behaving in the terms of its component processes. Traces, or sequential records of a process' behavior, allow observation of the events engaged in by a process. As the process starts, its path through all possible traces is not defined, though the complete set of all possible process traces can be known in advance. The choice will depend on environmental factors beyond the control of the process. This allows the modeling of the system in its environment, a key consideration for large complex systems.

Modechart. Modechart [Jah94], [Bro96], [Kar94] is a graphical specification language based on concurrent finite state diagrams. It provides a compact and structured way to represent real time systems. Although similar to Harel's Statecharts [Har87], Modechart is specifically designed for the specification of real-time systems.

Modechart is extended from Statecharts with constructs for expressing timing constraints. It has a visual hierarchical structure and a small set of well defined constructs for the definition of event-driven real time systems. These constructs include modes, mode-transitions, events, and timing constraints. During a non-zero moment in time, a mode can be either active or inactive. Informally, the state of a real time system is described by the collection of modes which are active.

Modes are hierarchically arranged in a tree structure; there is a top level mode from which all modes are descended and each mode has only one parent. The children of each mode can execute serially or in parallel, allowing Modechart to capture both concurrent and sequential behavior. The hierarchical nature of Modechart also allows specifications to be evaluated at different level of abstraction. a mode having no children is called an atomic mode.

Modecharts can be used to specify the timing properties of a system [Jah88]. The charts are built as a combination of modes (sets of system states in which some action or behavior will occur), conditions and events that cause the change of mode, and actions that will occur within each mode or during mode change. Modes can support hierarchy and thus the modeling of complex systems. At the same level of hierarchy, modes can be related to show either a single mode in operation or a set of modes in which the system must operate concurrently. The semantic of modes and transitions is expressed in real time logic (RTL), which distinguishes between action and event: while an action is an operation which requires a non-zero but bounded amount of system resources, an event marks a time value and requires no resources. An action can be executed while the system is operating in some modes or upon transition from one mode to another, and can be either periodic (occurring repeatedly at a defined rate within the sponsoring mode) or sporadic (occurring only when triggered by a specified event after the sponsoring mode is entered). The condition for mode transition can be either a triggering condition for taking the transition (condition for taking the transition depending on the occurrence of an event and/or the truth values of certain predicates), or a lower/upper bound. Modechart is an academic product without wide use in industry; its principal research thrust has been in the area of software development.

The behavior of a real time system is captured by mode transitions. Mode transitions are expressions which control the exit from one mode and entry into another. These transitions can be specified by timing constraints or can be triggered by events in the system or by predicates on the behavior of

modes in the specification. Events in Modechart include mode-entry, mode-exit events, transition events, and external events.

If M_1, and M_2 are modes, the exit of M_1 is indicated by $M_1\rightarrow$, the entry of M_2 is indicated by $\rightarrow M_2$, and the transition from M_1 to M_2 by $M_1\rightarrow M_2$. External events are represented by single letters, e.g. E. These form the atomic units of a system; assertions are described in terms of timing and ordering relationships between instances of these type of events.

Transitions can also be controlled by predicates on modes. For example, the predicate {(M)} indicates that the mode M is active at the current time period. Another useful predicate, {<M)}, indicates that the mode M has been active for at leas one unit of time. More precisely, M was entered before the current moment of time, and since then, M has not exited before the current moment Finally, more complex mode transition expressions can be formed from triggers and timing constraints. More elaborate triggers can be composed by taking the conjuncts of trigger expressions, and these conjuncts can be disjuncted together with timing expression.

Viewchart. The Viewcharts [Isa96] is also based on David Harel's Statecharts [Har87]. The Viewcharts notation extends Statecharts to include behavioral views and their compositions, limits the scope of broadcast communications, and consequently, reduces the complexity of scale that Statecharts faces in behavioral specifications of large systems.

The Viewcharts formalism can be viewed as a high-level specification language that uses Statecharts; and Statecharts is designed for real-time event-driven reactive systems. Furthermore, Viewcharts extends Statecharts to include behavioral views and their composition; consequently, it describes behavioral requirements of large-scale complex systems as compositions of views. The domain of the Viewcharts formalism, therefore, is behavioral specification of large-scale complex real-time event-driven reactive systems.

A behavioral view of a software system is the behavior of the system observable from a specific point of view. A client's view of a server, for example, is the behavior that the client expects from the server. This behavior, of course, may differ from the behavior that the server exhibits to another client. A server, therefore, may have several behavioral views. The caller view of a telephone set and the telephone set's view of a switching system are also examples of behavioral views.

Statecharts is often criticized for not being practical for large-scale systems specifications. Specifically, the issue of scale in Statecharts leads to the following problems;

- Explosion of States: In conventional finite state machines the number of states grows exponentially as the scale of the system grows linearly. Statecharts reduces this blow-up, but for a large-scale system this is still a problem.
- Global Name Space: All names in Statecharts are defined globally. A given event, for example, can be sensed throughout the system and, therefore, it must have a unique name. Managing the name space is difficult in large-scale systems.

Viewcharts attempts to eliminate these problems. A Viewchart consists of a hierarchical composition of views. The leaves of the hierarchy, described by independent statecharts, represent the behavioral views of the system or its components. The higher levels of the hierarchy are composed of the lower level views. Views are represented just like states, except that the arc-boxes representing views have thicker borders than those of states.

The Viewcharts notation limits the scope of broadcast communications. In other words, the scope of an element (event, action, or variable) in a given view is limited to the view. On the other hand, composition of views may require communication between the composing views; the scope of an event in one view, for example, may be extended to cover other views. In a given view, therefore, Viewcharts must distinguish two different types of events:

- Events that belong to (or are owned by) the view: These are the events that the view can trigger They must be declared by the view.
- Events that do not belong to the view: The view cannot trigger these events. An event of this type can occur only if it is triggered elsewhere and if the view is covered by the scope of the event.

An event may have multiple owners; in other words an event can be triggered by more than one view. An event may also have no owner, in which case the event can never occur. The Viewcharts notation allows this case, because of the possibility of further composition of the viewchart with additional views, which may affect the event. This is exactly analogous to the notation of free variables in program fragments, which can be bound in a larger context.

The notion of ownership for events is a natural consequence of composing views, while the scopes of events are limited. This notion also applies for actions. However, actions are implicitly declared, an action belongs to the view (or views) that generates (or generate) the action. There is no need, therefore, for explicit declaration of actions. However, an action may also be owned as an event by some other views, in which case it must be declared accordingly.

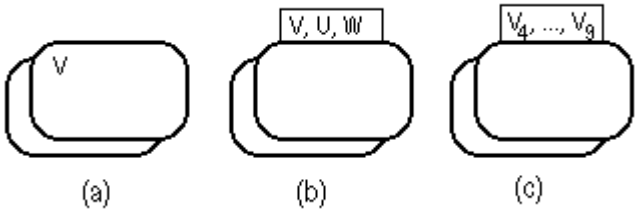

Figure 6.2.1.

Similarly, a variable belongs to the view that declares it. The scope of a variable declared by a view is the view and all its subviews. If a variable x is declared by a view V and redeclared by another view V1 within the scope of x, then Viewcharts recognizes two different variables which can be referenced by their qualified names, V.x and V1.x. In a view that is covered by the scopes of both variables, the base name x refers to V1.x. In the case of events, on the other hand, there is no need to specify them by their qualified names; Viewcharts determines the affect of each event occurrence based on the ownership and scoping rules. However, an event occurrence may still be specified by its qualified name, provided that it does not violate these rules.

Consequently, unlike events and actions, variables cannot have multiple owners. On the other hand, if a variable is used in a view, but not declared by the view or any of its superviews (i.e., it has no owner), then the variable, by default, belongs to the top view of the corresponding viewchart.

Syntactically, elements owned by a view can be declared by listing them following the name of the view either in the viewchart, as in Figure 6.2.2, or out of it as a separate text. In referencing a view by its name, however, it should be noted that the view must be uniquely identified. It may be necessary to identify a view but its fully or partially qualified name, which consists of the base name prefixed by the names of its ancestors in the hierarchy separated by dots.

In a viewchart, if the triggering view of an element is obvious and there is no ambiguity in the ownership of the element, then there is no need for explicit declaration of the element.

Composing behavioral views. Views can be composed in three ways: SEPARATE, OR, and AND compositions. Except for the effect of ownership and scoping restrictions, the OR and AND compositions of views, in Viewcharts, are similar to the OR and AND compositions of states, in Statecharts, respectively. The SEPARATE composition of views, however, is specific to Viewcharts. In addition, similar to the notion of depth in Statecharts, views can be encapsulated forming a HIERARCHICAL encapsulation of views.

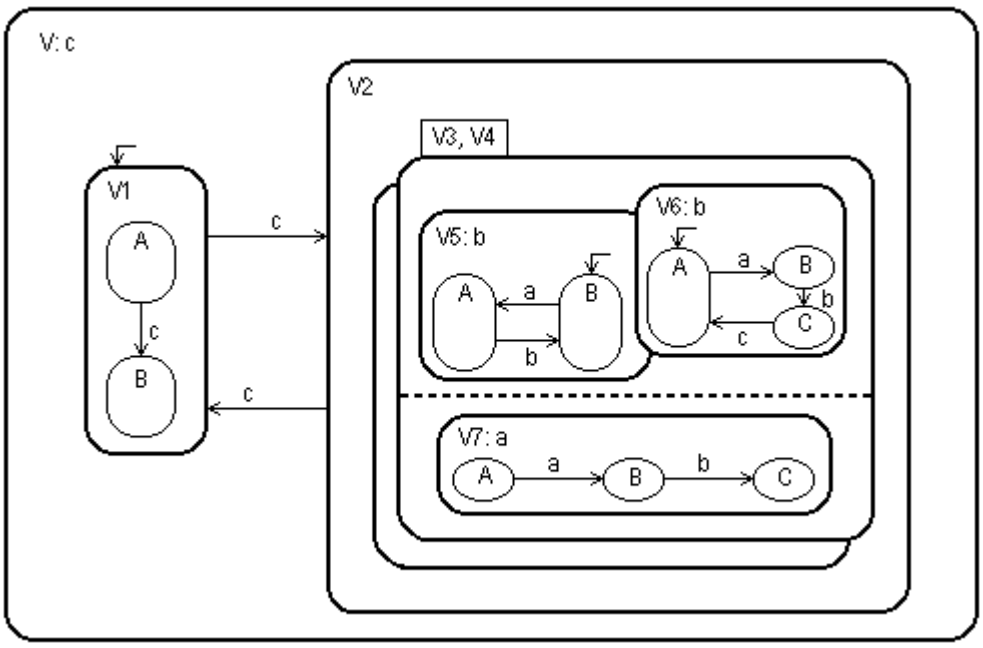

Figure 6.2.2. Viewchart

In a SEPARATE composition of views, all the views are active[1]; no transition between the views is allowed, the scopes of all the elements are unaffected; and any subview or state in one view is hidden from (i.e., cannot be referenced by) the other views. No view is, in fact, aware of any other view in the composition. Visually, the views involved in a SEPARATE composition are drawn on the top of each other, as shown in Figure 6.2.1, giving the impression that they are located on different planes and, consequently, are hidden from each other.

The representation (a), in this figure, specifies a SEPARATE composition of the view V with a finite number of other views; (b) specifies a SEPARATE composition of V, U, and W; and (c) specifies a SEPARATE composition of five views V4, ..., V9. In all these cases, the behavior of the first view, the one located on the top, can be specified. by default all the other views are identical to the first one. Exceptions are represented by specifying the other separately and referencing them in the composition by their names using the representations (b) or (c). If only a small number of views are involved in the composition, then it may be practical to give them enough space to show their behaviors. An example of this representation is given in Figure 6.2.2, which includes a SEPARATE composition of views V5 and V6.

The OR and SEPARATE compositions are similar, except that in an OR composition, only one view can be active and there can be transitions between the views. In Figure 6.2.2, for example, the view V consists of an OR composition of V1 and V2.

In an AND composition of views, all the views are active; the scopes of all the elements owned by each view are extended to the other views. All the subviews and states in one view are visible to (i.e., can be referenced by) the other views; variables, however, must be referenced by their qualified names. The view V7 of Figure 6.2.2, for example is ANDed with a SEPARATE composition of V5 and V6.

In a HIERARCHICAL encapsulation of views, some views form a superview; all the subviews and states in a superview are visible to the superview; and the scopes of the elements owned by a superview covers all its subviews.

[1] A view is active whenever the system is in a state of the view.

The viewchart of Figure 6.6.2, for example, is composed of a SEPARATE composition of V5 and V6, which in turn is ANDed with V7 forming V3. A SEPARATE composition of two identical views V3 and V4 forms V2. The full view V is an OR composition of V1 and V2.

Design Capture Views. A multi-domain design capture and analysis methodology has been described by Hoang [Hoa91] which partitions the system design capture activities into five Design Capture Views and defines relationships between those views in a formal manner. The five views partition the system design into intuitive segments corresponding to key perspectives of the system design. The Design Capture Views provide the systems engineer with a disciplined approach for capturing the following principal system design perspectives:

- Environmental Capture View addresses the operational scenarios, system concept of operations, environmental conditions, external systems and interfaces, system test strategies, maintenance and logistics considerations, and other external factors which impact the system under design,
- Informational Capture View captures a conceptual representation of the system under design employing entity-relationship diagrams and describes the system as a hierarchical Information Model,
- Functional Capture View establishes an implementation free traceable hierarchy (i.e. decomposition) of functions and related data flows, process specifications, and data descriptions for the system under design,
- Behavioral Capture View provides a mechanism for representing the real-time behavior of the system under design, and
- Implementation Capture View defines the physical hardware, software and human resources and resource architecture which comprise a particular embodiment of the system under design and its connectivity to external systems.

[Kar94] describes an advanced approach for capturing system behavior within a multi-domain system design capture and analysis methodology. This behavior capture approach addresses three key aspects of representing complex system behavior:

(1) response of the system to stimuli as observed externally (i.e. system as a black box),

(2) real time behavior of the system functional design options (i.e. control, timing, and synchronization aspects of the system functions and data without regard to specific implementation in resources), and

(3) real-time behavior of system implementation design options (i.e. response time, delays, queue lengths, and other aspects of the system behavior as embodied in hardware, software and human resource and resource architectures.

The first system behavior element is the top level system behavior as characterized by response of the system to external stimuli/events/conditions and to internally generated events/conditions. This element of system behavior treats the system as a black box and is concerned with describing the timing and response of the system under design in relationship to the external environment in which the system will operate.

Figure 6.2.3 illustrates the concept of top level system behavior. The top portion of Figure 6.2.3 illustrates the simplest case of top level system behavior where the system generates an action or event based upon a single external event or an internally generated event. The lower portion on Figure 6.2.3 illustrates a more complex case where a chain of multiple external and/or internal events are combined in a sequence with system generated responses. In each of these cases, the top level system behavior capture method must describe each external condition/event and each system response as well as the relationship of those events over time (e.g. the sequence) to adequately capture the behavior.

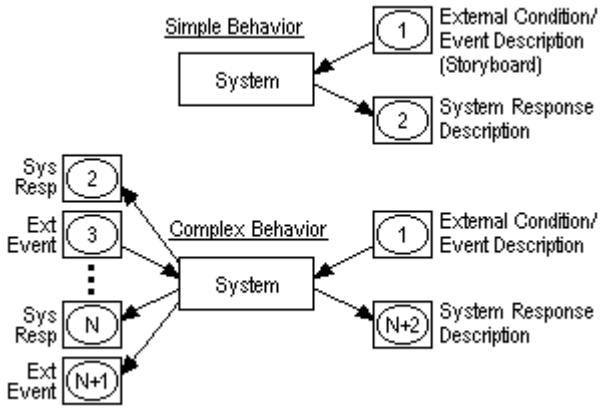

Figure 6.2.3.

An example of top level system behavior is the response time of the system to a user action such as a data base query. The initial external event which triggers this behavior may be represented by the entry of the query by a system operator. The system response would be represented by displaying the necessary information on the operator's display. The time associated with this top level system behavior of "system response time to data base query" may include a variety of components or different values. A required or budgeted value may be derived from the system top level requirements. A predicted or simulated value may be generated from an analysis of the system under design and a measured or observed value may be recorded from an actual test. A mechanism must be provided to capture and relate the spectrum of values which may be generated.

Top level system behavior is closely related to the descriptions of scenarios and test conditions captured in the Environmental capture view [Kar92]. Many top level system behaviors may be described in terms of a response to an external event. These external events are captured as part of scenario descriptions. A strong coupling between the descriptions of top level system behavior within the Behavior capture view and the scenario descriptions with the Environmental Capture View will therefore be explored as part of this task.

Both text-based and graphic-based mechanisms are being considered as candidates for capturing a description of top level system behavior. An innovative graphic story boarding technique which uses icons and symbols to represent various events and their relationship in time is a leading option. Other candidate capture techniques for representing to level system behavior include the behavior diagrams employed in the SYSREM methodology and implemented in the RDD-100 CASE tool.

The second element of system behavior addressed in this approach is the real-time behavior of the system functional design in terms of the control, timing, and synchronization aspects of the system functions and data (without regard to specific implementation in resources). This elements of system behavior is often necessary to completely specify the real-time aspects of the system logical design. Specification of data flow alone is not always sufficient to uniquely determine the timing and synchronization aspects of the system functions. This is particularly true when trading-off sequential vs. pipelined timing of system functions or operations as illustrated in Figure 6.2.4. The two examples in Figure 6.2.4 have identical functions and data flows however, the control structures are different. The control structure on the left implements sequential operation where in all functions in the chain operate in sequence on a given initial input before the next input is processed through the chain. The control structure on the right implements pipelined operation where in each function in the chain operates as soon as input from the previous function is available allowing multiple data elements to be processed chain simultaneously.

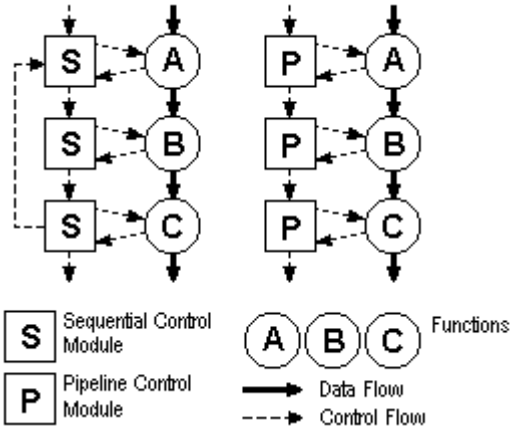

Figure 6.2.4

The control flow diagram explicitly specifies the sequence of operation for a set of system functions and is one of the candidate methods for capturing system functional behavior. Candidate capture techniques for representing system functional behavior in the form of control diagrams include RTSA, HGSDCSM, and SYSREM which are described in the preceding parts of this section. State Transition Diagrams (STDs) and other state based methods may also have potential to support capture of system functional behavior.

The system functional behavior is strongly related to the top level system behavior described above. The top level system behavior considers the system as a black box. The stimulus/response description at this level naturally serves as endpoints in tracing system functional behavior (which views the system as a white box) through the system as shown in Figure 6.2.5. The capture method for representing system functional behavior must provide a mechanism for relating to top level system behavior and tracing behavior through function/data flow threads in the functional design.

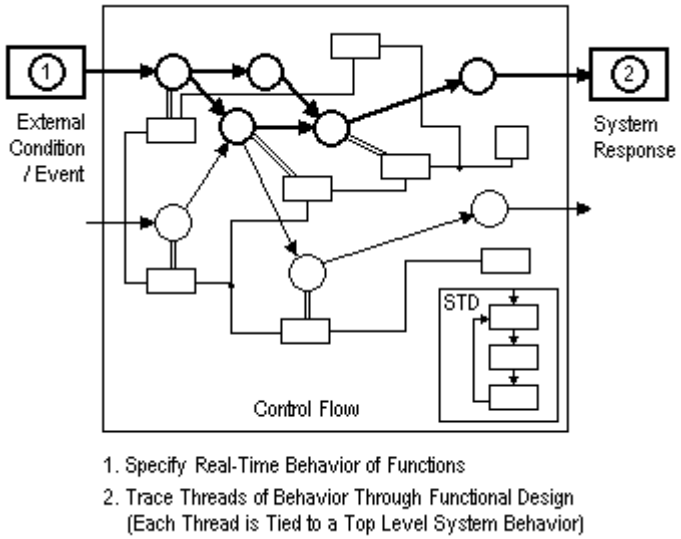

Figure 6.2.5

The third element of system behavior addressed in this approach is the real-time behavior of the system's implementation (or resource) design in terms of service time, delays, queue lengths, and other aspects of the system real-time behavior as embodied in hardware, software and human resources and resource architectures. The behavior of system hardware and software architectures is most often represented as selected testing threads through the system design. Each component in the path of a given thread contributes to the overall behavior of the system in terms of service time, delays, etc. This approach tends to isolate selected behavior issues for analysis and often can mask the effects of multiple concurrent threads (or contention for resources across threads). An extension

of this approach (which represents individual threads through an implementation) is to represent the real-time behavior of individual components and then to combine those components into a total system architecture representation. This extended approach still allows threads to be traced through the system implementation design and has the added capability to represent and analyze multiple concurrent system implementation threads.

As with the system functional behavior threads, system implementation behavior threads are strongly related to the top level system behavior described above. The stimulus/response description of the top level system behavior serves as endpoints in tracing system implementation behavior as illustrated in Figure 6.2.6. The capture method for representing system implementation behavior must provide a mechanism for relating to top level system behavior and for tracing behavior through hardware/software/human operator threads in the implementation design.

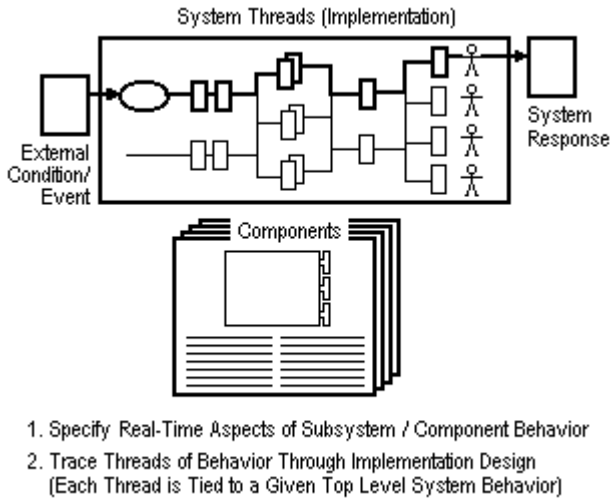

Figure 6.2.6

Automation Support for System Behavior Capture Methodology. Successful employment of an advanced system behavior capture and analysis methodology in supporting a complex system design is largely a function of the degree of mechanization which can be achieved. The size and complexity of large scale advanced computer systems render manual application of any design process or method unusable. Considerable potential benefits can be gained from automation which supports a disciplined structured capture of the initial iteration of a system design and subsequent editing of that capture. Further significant efficiencies can be gained through automated consistency and completeness checking within the behavior capture domain and between the other system capture views representing the captured design. However, in light of the overwhelming systems engineering task represented by analysis of an advanced complex system design, the most significant productivity gains may be in automated support for design simulation and analysis within an integrated and highly automated design capture and analysis environment.

Use Case Maps. The ability to attribute behavior to architecture is important for high-level understanding, designing, evolving, and reengineering all kinds of systems (from object-oriented programs to parallel and distributed computer systems). Scenarios are a good way of doing it, but popular scenario techniques, such as message sequence charts, that use intercomponent "wiring" as their starting point do not scale up well. Use case maps [Buh96] provide a new, scenario-based way of attributing behavior to architecture that solves the scaleup problem. The notation enables compact, composite maps to be drawn to represent behavior patterns of whole systems in terms of causal paths, without reference to "wiring".

Complexity Factors in Systems. Systems are often characterized at a high level of abstraction by block diagrams such as the following Figure 6.2.7 (called here component context diagrams), in

which the boxes represent collaborating components, every one of which is potentially-through recursive decomposition - itself a system.

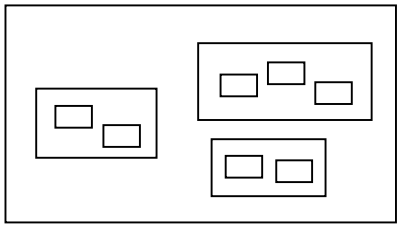

Figure 6.2.7

To attribute behavior to systems described with component context diagrams, connective tissue must be added. One form of connective tissue is intercomponent wires, meaning conceptual or actual connections that support interactions - such as calls or messages - between components through interfaces (a component context diagram with wires is called a wiring). Wirings are assumed to identify explicitly the names and parameters of all possible interactions, without reference to the internal logic of components. Behavior may he attributed to wirings by describing interaction scenarios with interaction diagrams [Jac92] (which we shall henceforth refer to by the generic term message sequence charts). We argue below that the combination of wirings and message sequence charts will overwhelm us with detail for systems of any size or complexity. We need to stand back from such detail to see the big picture clearly, in other words to see the system in architectural terms.

Furthermore, wirings that are concrete enough to be useful for understanding behavior in the above terms are too detailed to be considered as architecture diagrams unless systems are one-off, small, and simple. For families of large or complex systems, architecture should have more to do with rules or guidelines for creating concrete wirings over the family than with the specifies of individual wirings. Diagrams showing only the existence of relationships such as uses, communicates with, or has contract with are useful for architecture, but they are not wirings in the terms of this paper, and are therefore not starting points for attributing behavior to architecture.

Now let us examine the complexity factors associated with wirings and message sequence charts what will cause us to be overwhelmed with detail. They fall under three headings: operation, assembly, and manufacturing.

Operation. Wirings provide a means of understanding the operation of a system in terms of interactions over wires between components. Wirings must include wires not only between peer components at the same level of recursive decomposition but also between components at different levels of recursive decomposition, thorough all levels of decomposition (symbolized by the arrows in the Figure 6.2.8, remembering that there would be many more arrows in actual wirings). Such wirings give us a view of system operation in terms of compositions of may details, tending to overwhelm the big picture with details. The effect is compounded by two other factors.

Assembly. Systems may change form while they are running, in other words, the may be structurally dynamic (why we associate the term assembly with this will be explained shortly). Structural dynamics are a routine property of systems and software, not an unusual one. When systems are viewed in terms of wiring diagrams as above, structural dynamics mean that both the components and the wiring may change over time (symbolized in the Figure 6.2.9 by a sequence of snapshots, each of which show different components and different wiring at different times). We say that the wiring in each snapshot shows connections that enable operation, that sequences of snapshots imply assembly (meaning that new components and wiring must be assembled in between snapshots), and that operation + assembly = behavior.

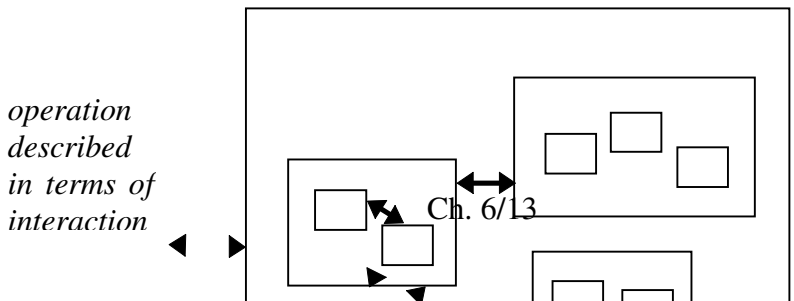

Figure 6.2.8

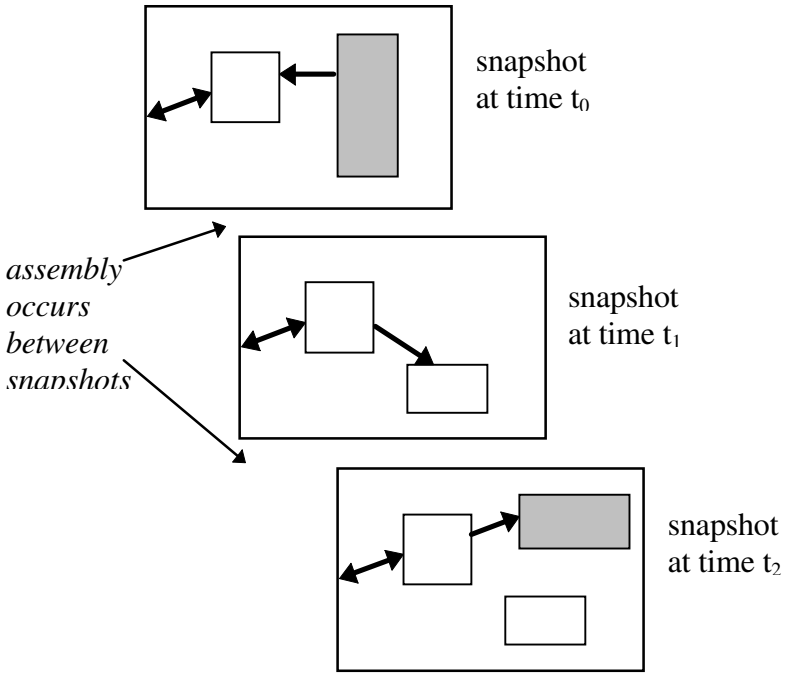

Figure 6.2.9

Even without any specific description of how assembly is to be accomplished in between the snapshots, viewing assembly in terms of changing wirings obviously adds complexity to an already complex picture. This complexity seems counter-intuitive, considering the lightweight way in which assembly happens in code (pointers are assigned for new software components and passed around to other components).

Manufacturing. The third complexity factor is that behavior (in the above sense operation + assembly becomes intertwined with manufacturing (in the sense of manufacturing objects from inheritance relationships in class hierarchies). This is symbolized in the Figure 6.2.10 by showing a class hierarchy on the right from which the operational components in the snapshots on the left are manufactured (the shaded shapes in the class diagram symbolize the idea that the inheritance hierarchy is, in effect, collapsed into concrete classes, instances of which become components in the snapshots). In object-oriented development, classes are where component behavior and intercomponent wiring are defined, but diagrams showing class relationships do not themselves give the system picture on the left, which must be inferred from details in the class descriptions. Thus the system picture is a second-class abstraction in relation to classes (second-class, because it depends on details of classes, namely their interfaces and methods). Many diagrams and snapshots may be required to see the whole picture.

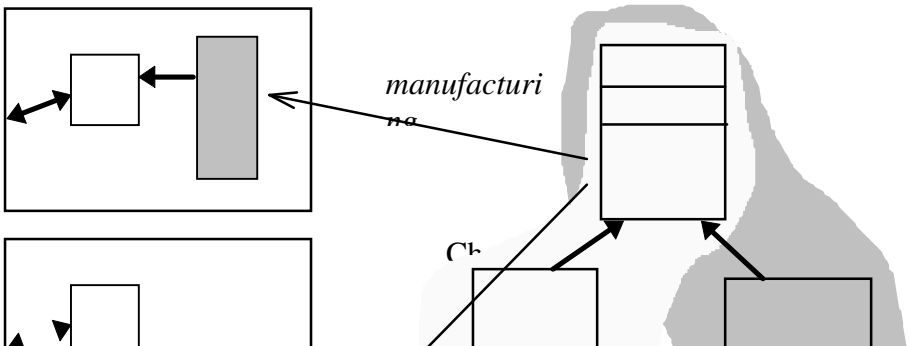

Figure 6.2.10

In Summary. Attributing behavior to systems through wiring drags the level of abstraction down below a level appropriate for architecture. It requires visually and mentally combining elements abstracted from a messy, difficult-to-separate soup of details. The approach may be tolerable for small systems, but does not scale up well.

Essence of Use Case Maps. Use case maps [Buh96a] raise the level of abstraction by simplifying all three of these complexity factors:

Operation. Wires are replaced as connective tissue by cause-effect paths. Behavior is not represented in terms of interactions between components vie wires, but in terms of cause-effect sequences between responsibilities of components. Wires to achieve the interactions required to implement the cause-effect sequences are deferred as details. In other words, interfaces of components, and interactions between interfaces, such as calls, messages, or IPC, are deferred as details. Responsibilities can be coarser grained than such quantities as calls or messages. The net effect is to greatly reduce the level of commitment to detail.

Assembly. The modeling of assembly as a sequence of snapshots of changing components and wirings is eliminated. The wiring part disappears because wires have disappeared. The changing components part is still required, but is modeled in a simple way through the use of slots. Slots are like positions in human organizations that exist independently of their occupants or occupancy. They are also like slots that exist in equipment racks of computer systems for circuit boards that may be plugged in at any time while the system is running. At the level of use case maps, the only requirement for puggability of a component into a slot is that the component is able to perform the responsibilities defined for the slot. Slots are not specifically illustrated in this brief description (see [Buh95], [Buh96a] for examples), but are an important feature of use case maps.

Manufacturing. Dependence of behavior descriptions on manufacturing details is eliminated by the fact that use case maps are first-class models in their own right that may be developed independently of class descriptions.

The Result. By eliminating dependence on wiring, use case maps raise the level of abstraction to an appropriate one for architecture.

The essence of use case maps is extremely simple. The Figure 6.2.11 is a use case map consisting of many paths superimposed on a component context diagram containing two components. The only thing to understand is that it is basically like a road map in which any end-to-end route (such as the one highlighted here with thick lines) may be traced with your finger to explain cause-effect sequences that follow from stimuli occurring at the start of the route (the filled circle). A use case map is basically a record of routes traced by one or more such finger-pointing sequences (called "scenarios"). Concurrence is represented, not by specific map notations, but by allowing multiple

scenarios to be traced at the same time, so use case maps are equally applicable to sequential or concurrent systems.

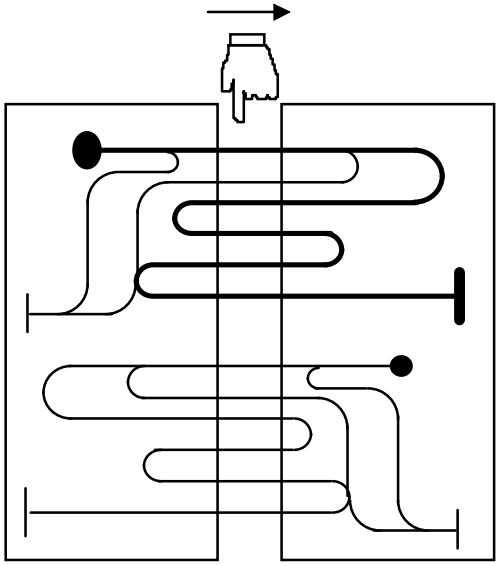

Figure 2.11

Use case maps have four main elements: *paths* that trace scenarios through the components of a system from points where stimuli occur to points where responses are felt; *responsibilities* that link paths to components; *couplings* between paths that express couplings between scenarios, such as one scenario pausing to wait for another; and *components* that perform responsibilities.

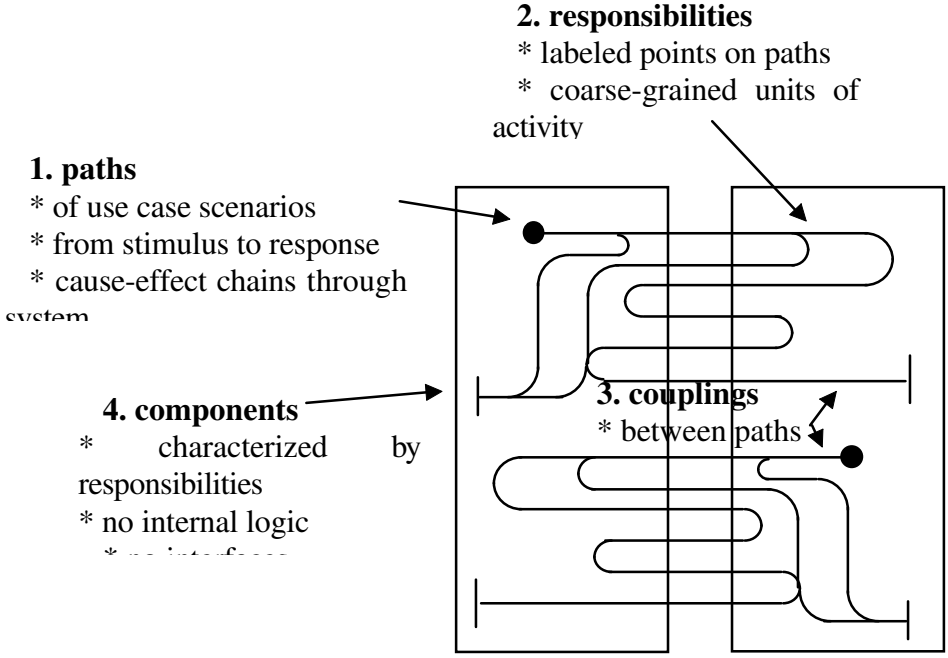

Figure 2.12

Paths. Use case maps provide a path-centric view (Figure 6.2.13) of systems that is, in a sense, a dual of the more familiar component-centric view.

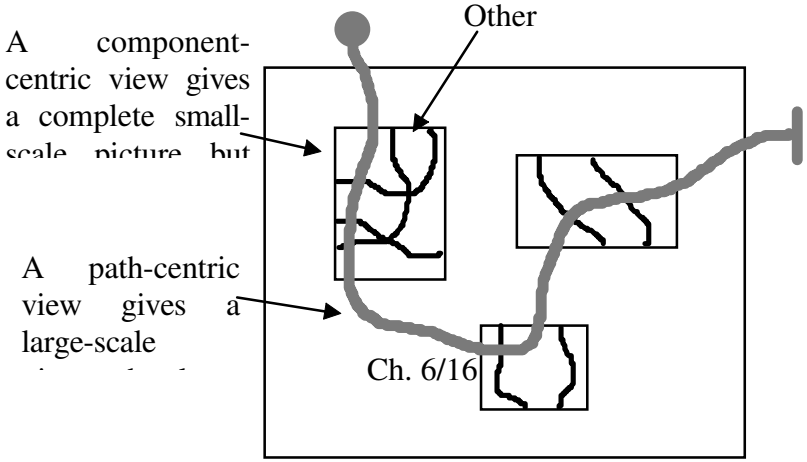

Figure 6.2.13

There is more to paths than just the visual representation. Documentation of maps includes at least the following: naming and prose descriptions of scenarios associated with paths; preconditions and postconditions; labeling of path segments to indicate the scenarios associated with them (e.g., A+B for a segment shared by scenarios A and B, to provide information that would be shown in diagrams - see Figure 6.2.14 - by shading or coloring); and cross references between large scale paths and factored smaller scale subpaths for parts of systems.

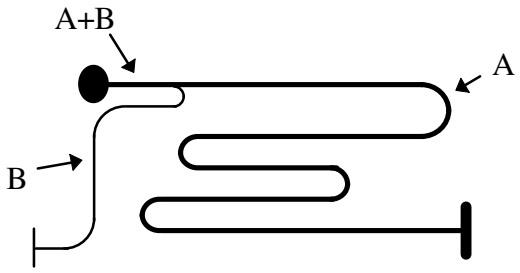

Figure 6.2.14.

Sequences of responsibilities are shown in use case maps as labeled points on paths over structure (Figure 6.2.15.), not as activity segments of timelines in separated time sequence diagrams. Thus they are not just a visual notation of responsibility sequences in use cases [Reg96], they are a direct link between use cases and architecture that combines both in one diagram. Many paths can be combined in a composite map to convey - in a compact way that is specifically related to architecture - information about behavior patterns extending over multiple scenarios. Thus maps exploit human pattern recognition capabilities to increase understanding.

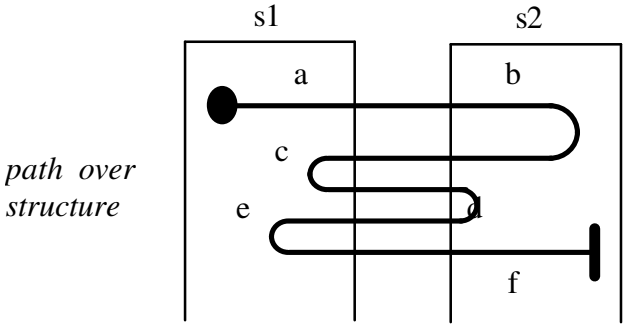

Figure 6.2.15

At the level of system architecture, we need to stand far enough back from fine-grained detail to see the big picture. This means that responsibilities may be coarse grained quantities, compared to calls or messages, and that sequences of responsibilities that span components may be coarser grained than sequences of calls or messages between components. Therefore, pairs of responsibilities that span pairs of components, such as ab, cd, or ef in the foregoing diagram, may actually represent situations in which multiple, detailed, back-and-forth interactions occur between the components to complete each pair of responsibilities. In other words we may have to think of such responsibilities as shared. The Figure 6.2.16 shows two ways of indicating shared responsibilities: as the ends of thickened responsibility segments stretching between components, or with a special syntax for the responsibility.

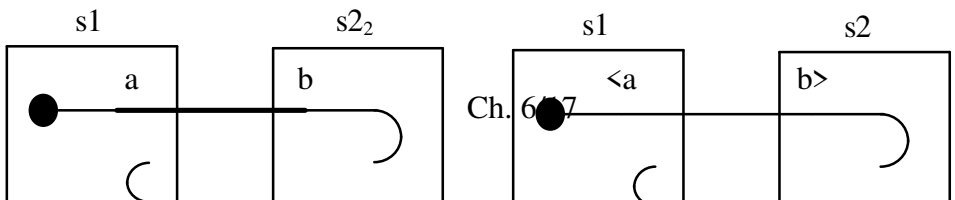

Figure 6.2.16

Components include fixed components (indicated by solid outlines) and slots (indicated by dashed outlines - not illustrated). Slots enable assembly to be expressed with fixed use case maps in as lightweight a fashion as it is implemented in code, without leaning on programming concepts. A simple notation indicates points along paths where components are created, moved, or destroyed. At points of movement, the movement is into or out of slots or pools (a slot is a place where a single component may ba active, a pool is a place where many components may be held in readiness to move into slots). Paths between such points indicate sources and destinations, without indicating the transport mechanism.

In fact, in software, components do not actually move, they only change visibility. Saying a component moves from one place to another is just another way of saying it goes from being visible in one place to being visible in another, where "visible" means it can be interacted with. From a software perspective, the movement model is a convenient fiction that contributes to a lightweight representation of assembly in use case maps.

Interpath coupling may be implied by scenario patterns or indicated explicitly by coupling notations. In either case, the existence of coupling is visible in the maps, it is only a question of whether or not there are explicit coupling notations joining paths. Examples of coupling implied by scenario patterns are as follows: the postconditions of one path become the preconditions of another; paths share some physical resource that implicitly couples them (e.g., a communication channel); and a responsibility along one path, by definition, affects a responsibility on another path through the state of some shared component.

6.3. Modelling and Programming

6.3.1. Real-Time ANSA.

This section describes the real-time programming model of real-time ANSA based on [Li94]. Its scheduling flexibility will be demonstrated by its two-level scheduling multiplexing. Policy/mechanism separation is used to address the diversity of real-time programming. An integrated priority management scheme will be introduced for preemption control. The real-time programming model provides a framework to facilitate the enforcement of stringent timing constraints found in distributed real-time applications.

ANSA. The ANSA architecture is based on the distributed object (client/server) model. Services are provided at interfaces which are typed and a conformance relationship is defined to allow one interface to be substituted for another, functionally equivalent interface. Interfaces contain operations which are invoked to make use of the service offered by that interface, thus clients make use of services by invoking operations in the interface(s) provided by the service.

Distributed Object Execution. The use of an object-oriented data model and the client-server execution model makes the distribution of data and the processing implicit in nature. In non-real-time

environments, object-oriented design has been successful in simplifying the design, implementation, and maintenance of software in many distributed systems.

Object interdependence can be classified into two categories: static interdependence - the structural relationships between objects, and dynamic interdependence - the interactions between objects. Many useful results are known about the static relationships between distributed objects. Related concepts, such as abstract data typing, type checking and subtyping, are accepted and used widely. On the other hand, little consensus has been achieved on the execution view of objects.

For real-time applications, this execution aspect is of vital importance - it has a fundamental impact on the predictability of computational activities. Real-time object execution models are required to address not only how the computational activities are carried out, but also how shared resources are used (i.e. the manner in which contention for system resources is resolved taking into account timing constraints of real-time activities). Distributed real-time systems must provide-support for the specialized requirements of real-time communication, tasking, scheduling, and control. These requirements must be explicitly addressed in an object execution model, if the object oriented approach is expected to be applicable to a real-time world.

Designing a Real-Time Programming Model. The essence of a real-time programming model is to provide the basic abstractions such that stringent timing constraints of real-time activities are respected (guaranteed at best). The real-time programming model described in this section is based on the ANSA computation and engineering models. As in the ANSA system, objects provide the basis for distribution, interfaces of objects provide service access points, and named operations of an interface provide the actual services. Abstractions, mechanisms and policies are developed to allow a programmer to access and control the resource allocation of the supporting environment. Tasks (representing processor resources) and communication-channels (representing communication resources) are considered the most important system resources. Both static resource allocation - the allocation of system resources to interfaces - and dynamic resource allocation - the allocation of system resources to invocations are supported. Predictability, user control and mission criticality are the main concerns of the real time programming model.

6.3.2. A Real-Time Programming Model.

This section discusses the real-time aspects of real-time ANSA objects. The structure of the real-time objects is examined along with object invocation mechanisms, the handling of priorities and deadlines, resource allocation, scheduling mechanisms and policies, and the application's control over scheduling.

ANSA object execution. The ANSA object execution model is defined by the ANSA Computation Model and ANSA Engineering Model, which are similar to the ODP computational and engineering viewpoints [ISO92]. The ANSA Object Execution Model can be summarized as follows.

- objects export services through interfaces.
- threads are created either explicitly for concurrent computational activities or implicitly by the invocations between objects. In the latter case, a thread embodies a distinct run-time agent for a client in its server side, representing the invocation on a computational interface.
- the infrastructure (capsule) is in charge of the management of resources (tasks[2], buffers etc.) in the system, and of their allocation to the different threads.

This means the system behavior is completely dependent on the system's resource management policy. Also, the infrastructure offers no possibility of interacting with this management. Therefore,

---

[2] A distinction of thread and task is made in ANSA. A thread is a sequence of instructions representing a unit of potentially concurrent activity. A task is a virtual processor which provides a thread with the resources (e.g. a stack) it requires to progress. From engineering point of view, threads are cheap resources (each requires less than one hundred bytes of memory); whereas tasks are expensive resources (each requires several kilo-bytes of memory).

the resulting behavior is totally non-deterministic, and nothing can be guaranteed; it depends entirely on the system workload.

Real-time objects. The real-time object model is inherited from the ANSA object model, with extensions to provide resource allocation and real-time scheduling support.

Like ANSA objects, a real-time object is composed of data, one or more tasks of execution, and a set of exported interfaces. A new abstraction, entry, is introduced as the basic mechanism for real-time scheduling. An entry is a thread queue with a record of control data. An entry may be created dynamically, and interfaces of an object may be bound to it. When an interface is bound to an entry, each operation request on the interface will be transferred (by the infrastructure) to a thread enqueued on the entry. Any thread representing a computational activity is also spawned on an entry. The entry is an engineering concept which is confined within a capsule.

In Figure 6.3.1 a graphical illustration of a real-time object is given.

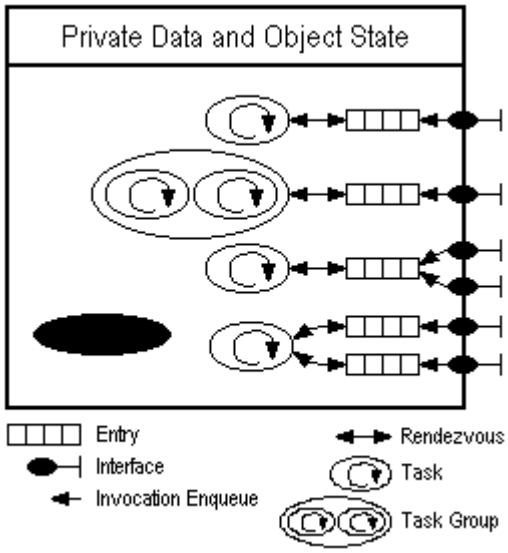

Figure 6.3.1

Flexible tasking is based on the entry abstraction. System tasks may be allocated for each individual entry. The tasks allocated are dedicated to execute the threads on the entry. When executing a thread, a tasks is also allowed to rendezvous with other entries dynamically. A rendezvous of a task with an entry means that the task waits to accept and execute one thread on the entry. Different control parameters may be selected for each entry to choose a thread enqueue policy, a task/entry rendezvous policy, and to enforce concurrence controls.

In such an object model with data, interface, entry and tasks encapsulated within a capsule, there is a choice of how many entries are allocated, which interface is attached to which entry, how many tasks are allocated to an entry, whether a task can rendezvous with a specific entry, and what kinds of resource scheduling policies are used.

The choice to allocate a new entry for some interfaces reflects the need to separate these interfaces from others for the purpose of resource management. The number of tasks allocated to an entry not only enforces the real concurrence allowed for the execution of threads on the entry, but also affects the real-time scheduling properties, for example, preemptivity.

The flexibility for allowing a task to rendezvous with an entry enables an application to have complete control over its virtual processor(s) based on its knowledge of the system state. The user control over system tasking behavior is further enhanced by the scheduling policy/mechanism separation used in the architecture. These resource management activities can all be done dynamically, increasing the flexibility and usefulness in an open dynamic environment.

Real-time object invocation. The act of requesting that an operation of an interface be executed is termed an invocation. Each invocation is conveyed as a message to the invoked object, and is then transferred to a thread in the capsule where the invoked object resides.

To support the mission-critical requirements, there must be some means to enable the urgency of a computational activity to be spread among all the modes it needs to access; and that urgency information should be used by the system resource scheduler to resolve resource contention so that important or more urgent computational activities have better access to system resources. This is done in real-time ANSA by allowing the association of an optional priority and/or deadline with each invocation. As the invocation crosses the physical boundary and becomes a thread in the called object, this priority and/or deadline is also passed and becomes a property of the thread, which may then be used as a scheduling parameter on the server site.

It should be pointed out that the priority and/or deadline is just a client's objective view of the criticality of an invocation; how that will affect the system resource management is also determined by the scheduling policy (the interpretation of the scheduling parameters) and the resources allocated for the service. This is further explained in the following sections.

Scheduling. The main goal of the real-time ANSA tasking design is to allow the maximum control of scheduling at the application level. Care has been taken to achieve a balance between flexible and deterministic scheduling. A policy/mechanism separation approach has been taken to address the diversity of real-time programming. Real-Time programming models have been devised for specific applications. Therefore, an ideal general purpose real-time support environment should provide multiple models of real-time programming. This is supported by the multiple application-selectable scheduling policy modules on top of a shared set of scheduling mechanisms.

The system scheduling behavior is defined in layers as:

- thread scheduling - the rendezvous scheduler on each entry.
- task scheduling - the nucleus scheduler on tasks.

Task scheduling and thread scheduling are two separate, but related, scheduling domains. Task scheduling is defined in the nucleus or the underlying operating system kernel. Thread scheduling is defined per entry. Task scheduling manages the multiplexing of task executions over processor(s). Thread scheduling manages the multiplexing of requests (threads) over tasks.

The primary function performed by multiplexing is the sharing of processor resources, which is similar to the multiplexing in communications systems and protocols for sharing communication resource. The use of separate entries to process requests on separate interfaces offers a number of (potential) advantages:

- allows the use of a specific scheduling policy (thread scheduling policy) suitable for each interface or each interface class.
- allows the possibility of using interface specific tasks to serve requests, and thus allows for more efficient resource utilization.
- separate entries may be processes in parallel, thus increasing performance.
- allows the possibility of end-to-end scheduling and guarantees.
- preserves the modularity and separation of service interfaces.

The nucleus scheduler defines how the real processor(s) is assigned to tasks, i.e. it manages the context switches between tasks. Preemption is used together with task scheduling parameters to order (either partially or completely) the otherwise non-deterministic behavior of the task execution.

There are two issues in thread scheduling management. One is how a thread is enqueued in an entry (with the assumption that the first thread in the queue is executed first). Such a policy may be a

system-defined one (such as invocation priority based or invocation deadline based) or an application provided one. Another issue is how a serving task rendezvous with a thread in an entry, i.e. how the thread scheduling parameters (priority and/or deadline) are used/inherited by the task. This is defined by a task/thread rendezvous policy. Such a policy affects how the serving task competes for processor resources with other tasks.

Priority scheduling. This section discusses the mechanisms needed to provide the static priority based scheduling model in the real-time ANSA framework. Though obviously related, priority and scheduling are different issues. Associating a notion of priority with an invocation is an intuitive way of structuring real-time applications.

The priority queuing of threads in an entry is incorporated to support such a view of real-time applications. While priority is a well defined and generally applicable notion, its role in task scheduling needs to be carefully examined. A clear definition of the priority inheritance and priority ceiling - used when the enforced synchronization during a task and a thread rendezvous - is needed to understand how priority works on tasking.

Priority management and priority inheritance. A distinction is made between a task's static priority (that declared in its creation) and its dynamic priority (that is the static value potentially enhanced by a rendezvous or an explicit change of priority). it is the dynamic priority that is used by the nucleus (or operating system) schedule to determine the current system-wide urgency of a task.

The tasking model is designed to support a structured approach to priority management. Statically the different task/entry/interface configurations allow important real-time services to be distinguished from non-real-time services. A dedicated entry may be allocated to real-time services, and high priority tasks may be allocated on the entry, so that a request on the interface has better response time. Dynamically, a serving task may take into account the priority of an invocation, and use this priority as its dynamic priority. This a called priority inheritance.

Two levels of priority inheritance schemes may be defined. They are called (basic) priority inheritance and transitive priority inheritance. In the first scheme, a serving task with a low priority raises its priority to the higher priority of an invocation request before it states the service, and changes back to its original value after the service is completed. The second scheme is an extension of the first scheme to consider the situation when there are no waiting serving tasks and a high priority invocation request arrives. In this case, the invocation priority is compared with the priorities of the running serving tasks. If all of the serving tasks are running at priorities lower than the invocation priorities, one of the tasks is chosen to inherit the invocation priority. If at least one of the serving tasks is running at a priority which is higher than the invocation priority, then the invocation is enqueued in the entry.

Transitive priority inheritance is difficult to implement[3]. An alternative approach is priority ceiling. Each entry may be associated with a fixed priority ceiling value, which specifies an upper-bound priority that applies to all the invocations on the interfaces bound to the entry. While a task is executing a thread on the entry, its priority is raised to the ceiling priority. If an invocation has a higher priority than the ceiling priority, it is rejected. Priority ceiling is easy to implement, but may introduce some unnecessary blocks.

Other scheduling paradigm. A deadline value associated with an invocation specifies a bound on the completion time of the requested operation. By assigning deadline values to invocations, the problem of satisfying timing constraints becomes one of scheduling processes to meet deadlines, or deadline scheduling.

A simple deadline scheduling policy is to treat deadlines as priorities in thread queuing. An earlier deadline has higher priority than a late one. Let's call it deadline based thread scheduling. It is not

[3] To implement transitive priority inheritance, the infrastructure needs to maintain the dynamic task/thread relations and requires special kernel supports of transitive priority inheritance operations.

assumed that the task scheduler understands deadlines. The resultant behavior is a non-preemptive earliest deadline first execution of invocations.

Preemption is possible if the task scheduler provides an earliest deadline first preemptive scheduling service and serving tasks are allowed to inherit thread deadlines. Under these conditions, deadlines can be handled exactly as priorities as defined in the last section. It should be pointed out that deadline based scheduling provides only a deterministic scheduling approach. It provides no guarantees for satisfying deadlines.

As deadlines impose timing constraints directly to invocations, a late result produced by a server task has little or no meaning. This timeliness requirement suggests that the RPC protocol - the Remote EX-execution protocol in the ANSA system - should take deadlines into account.

Priority and deadline scheduling can also be combined to provide alternative scheduling models.

6.3.3. ROOM - Real-Time Object-Oriented Modeling

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.

Different application domains require different architectural forms [Kog94]. 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.

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.

In this section we describe a practical approach for specifying the architectures of distributed real-time systems based on [Sel96] and [Sel94]. Architectures are explicit, formally specified, end enforceable. This approach uses an object-oriented, high-level specification language called ROOM (an acronym for **R**eal-time **O**bject-**O**riented **M**odeling) which can be used to automatically derive implementations.

The ROOM Language. 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 change-ability, 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.

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 aspects 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 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, these are relegated to the status of project folklore, treated by the implementation teams as the outdated and oversimplified products of ivory-tower naiveté.

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 section, 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.

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 level of a system, form the architectural down to the finest implementation detail. This, however, creates the problem of separating the "forest" (i.e., the architecture) form 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 (Figure 6.3.2), called the Schematic Level (due to its graphical syntax), is used to specify the higher scopes of as 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.

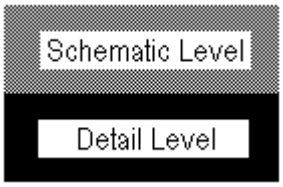

Figure 6.3.2

To make the architectural specification more easily discernible, ROOM uses a graphical syntax at the Schematic Level 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" object 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 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).

The Communication Model. In ROOM, the occurrence of an event is signaled 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 in Figure 6.3.3).

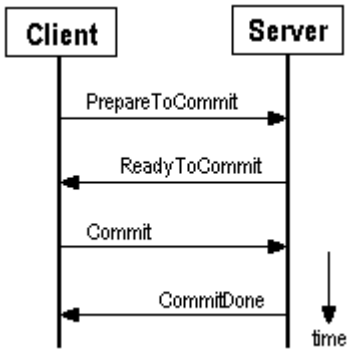

Figure6.3.3

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 do 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 is early architectural work where the full details of an object collaboration are either irrelevant or unknown.

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.

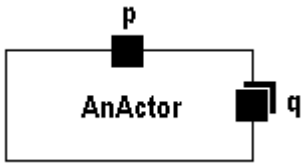

Figure 6.3.4

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, they 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 Figure 6.3.4 is a replicated port (indicated by the extra flange on the port icon). Actor instances can also be replicated if they are all instances of the same class (Figure 6.3.5).

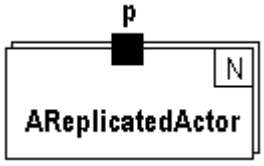

Figure 6.3.5

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 communications 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 (Figure 6.3.6) indicates that these ports use an "inverse" of the protocol class at the other end of the binding.

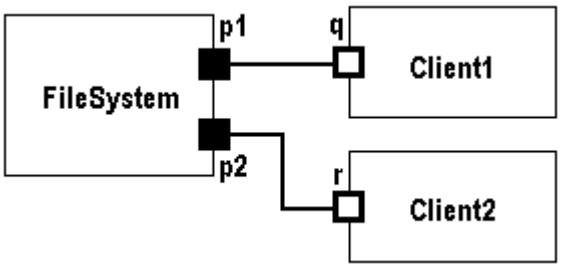

Figure 6.3.6

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 (Figure 6.3.7).

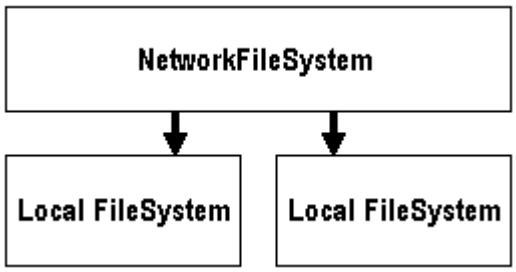

Figure 6.3.7

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 connection 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 its should be possible for actors to be decomposed into aggregates of more elementary actors. For example, the FileSystem actor form the Figure 6.3.6 might be decomposed as follows.

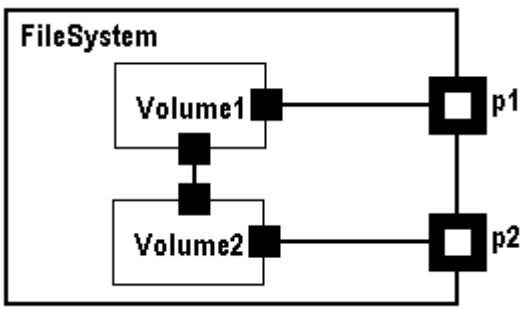

Figure. 6.3.8.

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.

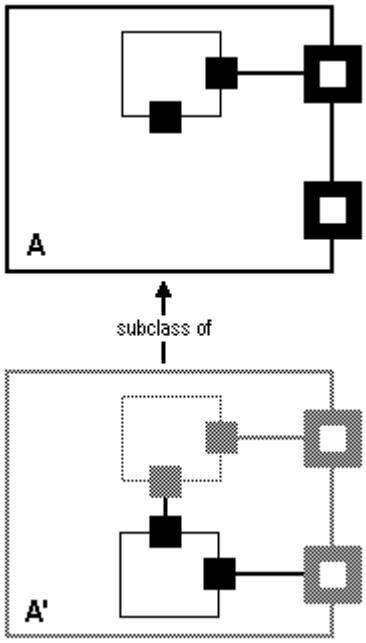

Figure 6.3.9.

Note that the two ports on Figure 6.3.8, 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 it 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 Figure 6.3.9. In the subclass, all inherited attributes are drawn 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 (Figure 6.3.10).

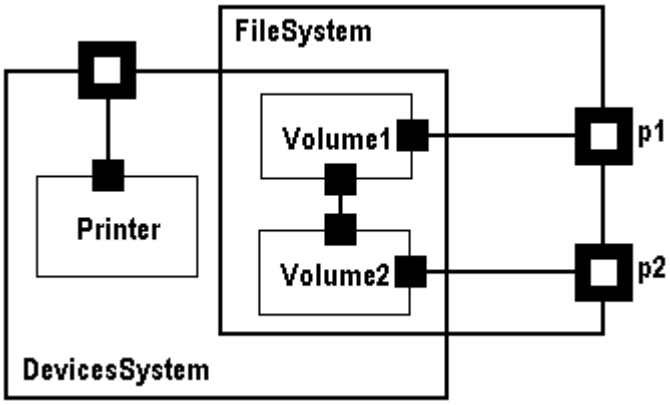

Figure 6.3.10

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 intentional 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 in Figure 6.3.11. 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.

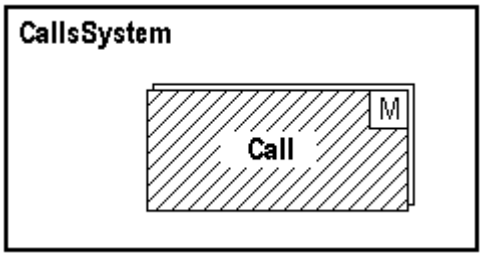

Figure 6.3.11

Assume that a simple telephone call is represented by a composite actor that contains the two subscriber actors involved in the call (Figure 6.3.12).

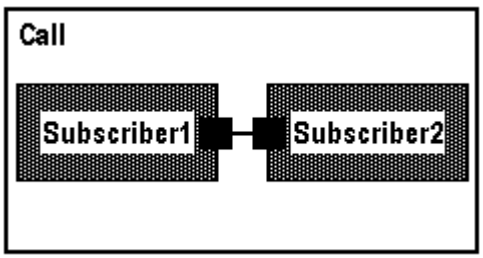

Figure 6.3.7

When defining this actor class, we cannot known 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 "lots" into which specific instances will be placed at some point. Such

place-holder 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.

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 (Figure 6.3.13).

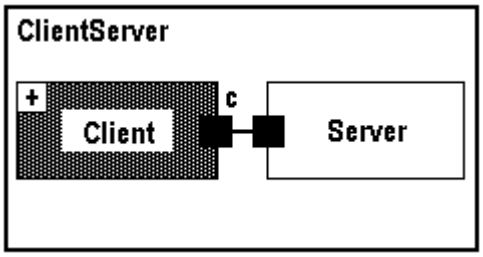

Figure 6.3.8

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.

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 massage 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 hierarchical 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[4] (this implies queuing of messages at the interfaces). The advantage of this approach is that it provides automatic mutual exclusion of event handlers and,

[4]However, other actors may process their messages during that time.

thereby, significantly simplifies the behavior specification. It is justified in situations here 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 in Figure 6.3.14).

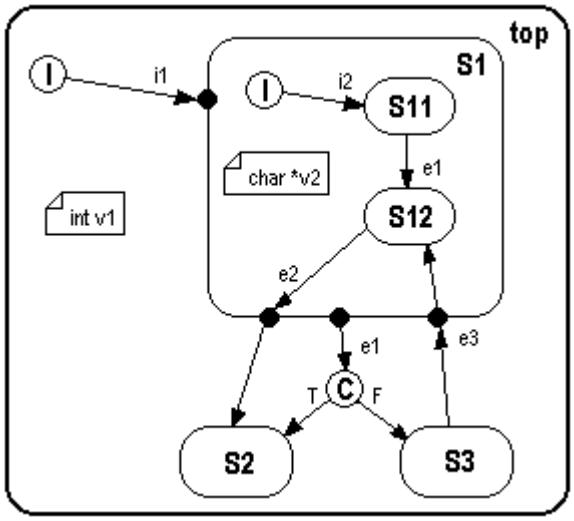

Figure 6.3.9

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 [Har87]. For example, transition e1 in the ROOMchart (Figure 6.3.14) 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.

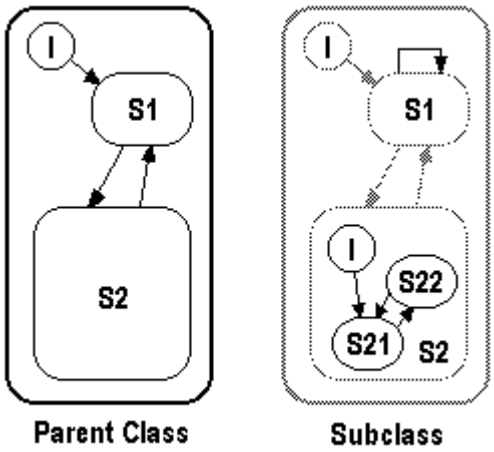

Figure 6.3.10

In this case, a subclass automatically inherits all the behavior attributes of the parent class and can refine that behavior by adding further attributes (Figure 6.3.15). 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.

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 to 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 [Har87]. This is because the software realization of concurrent states often requires expensive and inefficient mechanisms (including open-ended broadcasting and synchronous execution).

Code generation from ROOM models in 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[5]. 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

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

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 selection 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 he 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 stretch. 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.

6.3.4. JAVA

Java is a new programming language publicly released by Sun Microsystems in May of 1995 with the intention of revolutionizing the software industry. The popular press has responded with numerous articles touting the language's advantages over available alternatives. Since many of the applications which Java is intended to serve have real-time characteristics, some research teams have recently undertaken to develop a set of standard extensions to provide Java programmers with the ability to describe the real-time requirements of their Java applications. This section summarizes the issues that have influenced the design of Real-Time Java and provides an overview of its current embodiment based on [Nil97].

Java Background. Development of Java began in April of 1991. Initially, Java was intended to be an implementation language for personal digital assistants. Subsequently, the development effort was retargeted to the needs of set-top boxes, CD-ROM software, and ultimately the World-Wide Web. It is as a medium for portable distribution of software over the internet that Java is currently attracting all of its attention. However, both within and outside of Sun, it is well understood that Java is much more than simply a language for adding animations to Web pages.

Java is an object-oriented programming language with syntax derived from C and C++. However, Java's designers chose not to pursue full compatibility with C and C++ because they preferred to

eliminate from these languages certain troublesome features. In particular, Java does not support enumerated constants; pointer arithmetic; traditional functions, structures and unions; multiple inheritance; goto statements; operator overloading; and preprocessor directives. In their place, Java requires all constant identifiers and functions (methods) to be encapsulated within class declarations. Java provides standardized support for multiple threads (lightweight tasks) and automatic garbage collection of dynamically allocated memory. Furthermore, Java fully specifies the behavior of every operator on every type, unlike C and C++ which leave many behaviors to be implementation dependent. These changes were designed to improve software scalability, reduce software development and maintenance costs, and to achieve full portability of Java software.

One distinguishing characteristics of Java is its execution model. Java programs are first translated into a fully portable standard byte code representation. The byte code is then available for execution on any Java virtual machine. A Java virtual machine is simply any software system that is capable of understanding and executing the standard Java byte code representation. To prevent viruses from being introduced into a computer by a foreign Java byte-code program, the Java virtual machine includes a Java byte code analyzer that verifies the byte code does not contain requests that would compromise the local system. By convention, this byte code analyzer is applied to every Java program before it is executed. Byte code analysis is combined with optional run-time restrictions on access to the local file system for even greater security. Current Java implementations use interpreters to execute the byte codes but future high-performance Java systems will have the capability of translating byte codes to native machine code on the fly. In theory, this will allow Java programs to run approximately at the same speed as C++.

Java Overhead. Java is a high-level language, offering run-time support for dynamic binding of method invocations (function calls) and automatic garbage collection. There are certain costs associated with these high-level services.

To estimate the garbage collection costs of Java, we refer to recently reported studies of similar languages. In reference [Det94], the overhead of conservative mark-and-sweep garbage collection of eleven C and C++ programs was reported to range from 14 to 69% of each program's total execution time. In comparison with four explicit memory management techniques, the garbage collector performance was on average 4.6 to 11.1 times slower. Accurate generational garbage collection has not yet been measured for Java-like workloads. It is important to recognize that real-time garbage collectors typically have higher overheads than stop-and-wait or generational techniques [Sch94]. There are two reasons for this:

(1) in order to make sure that background garbage collection activities do not interfere with the ongoing execution of real-time tasks, the total garbage collection effort is divided into many very small incremental steps, necessitating frequent performance robbing context switches; and

(2) in limited memory embedded systems, it is necessary for the garbage collector to recycle memory much more aggressively than is typical of non-real-time systems. Otherwise, the application code would run out of dynamic memory before its supply could be replenished.

Taking all of these factors into consideration, we expect that typical Real-Time Java programs will spend between 30 and 50% of their execution time performing work directly related to garbage collection. However, we point out that hardware-accelerated garbage collection techniques that are currently under development make it possible to reduce the overhead of real-time garbage collection to less than 10% of the system's total execution time for many representative workloads [Nil95].

For large applications, Real-Time Java's dynamic memory utilization efficiency is likely to be slightly superior to the efficiency of most available alternatives for explicit real-time management of dynamic memory [Nil95]. However, utilization of dynamic memory is generally much less efficient than utilization of static memory. This is because dynamically allocated objects are more constrained by alignment restrictions and are accompanied by run-time tags and bookkeeping information. Based on several years of studying dynamic memory allocation for embedded real-time applications, it can be

estimated that a carefully hand-constructed real-time system requiring up to 1 Mbyte of RAM would be equivalent in capability to a Real-Time Java system requiring approximately 4 Mbytes of RAM. It is probably not wise to attempt to build a Real-Time Java system with anything less than 4 Mbytes. And the benefits of hand optimization of memory usage decline for systems larger than 1 Mbyte. As systems grow in complexity, hand-tuned memory allocation is increasingly problematic, often resulting in storage leaks, dangling pointers, and long-lingering bugs in application software.

For systems requiring 16 - 64 Mbytes of physical RAM, it can be predicted that Real-Time Java implementations will on average offer approximately the same memory utilization efficiency as would be expected for a C or C++ implementation of equivalent functionality. For systems requiring even more memory, Real-Time Java implementations will likely offer superior memory utilization performance. This is because the statistical performance of automatic garbage collection is known to be superior to general purpose manual memory allocation techniques. But for small systems, specialized hand-tuned techniques offer better memory utilization than garbage collectors (at the cost of requiring additional programming effort). And for slightly larger small systems, the fixed operating overheads of implementing garbage collection are likely to cost more than the efficiency benefits that are achieved through the use of improved memory allocation algorithms. Of course, claims such as these are difficult to prove: once a particular implementation technique has been demonstrated to be adequate, it is difficult to motivate anyone to reimplement a large complex system using what are suspected to be inferior techniques just for the sake of testing the hypothesis.

Another source of overhead that prevents Java from running as fast as C is the cost of implementing dynamic dispatch. In object-oriented programming languages, function (method) invocations are more common than in typical imperative languages like C. Furthermore, the cost of invoking a Java method is greater than a function call in C because the code associated with the method must be determined at run time, depending on the type of the object to which the method is to be applied. Though these costs have not yet been rigorously studied for Java implementations, we would expect its performance to be similar to that of C++. One study of a compile-time optimization technique that replaces dynamic method lookups with static calls and in-lined method bodies demonstrated a median 20% performance improvement for a set of 7 realistic C++ programs [Aig95]. Since this optimization replaces a median of 80% of all virtual function calls, the performance cost of dynamic method dispatch in comparison with C is a minimum of 20% of execution time, at least for programs that are written in typical object-oriented style.

In summary, embedded applications written in Java are likely to require faster processors and more memory than carefully hand-crafted C implementations. However, for larger systems, it is less practical to dedicate large amounts of programmer effort to hand optimization. Java's benefits are most valuable for implementations of these larger systems.

Real-Time Challenges. Unlike most existing real-time systems, many of the applications for which Real-Time Java is intended are highly dynamic [Nil95a]. New real-time workloads arrive continually and must be integrated into the existing workload. This requires dynamic management of memory and on-the-fly schedulability analysis. Price and performance issues are very important, making certain traditional real-time methodologies cost prohibitive. An additional complication is that an application developer is not able to test the software in each environment in which it is expected to run. The same Java byte-code application would have to run within the same real-time constraints on a 50 MHz 486 and on a 300 MHz Digital Alpha. Furthermore, each execution environment is likely to have a different mix of competing applications with which this application code must contend for CPU and memory resources. Finally, every Java byte-code program is supposed to run on every Java virtual machine, even a virtual machine that is running as one of many tasks executing on a time-sharing host. Clearly, time-shared virtual machines are not able to offer the same real-time predictability as a specially designed real-time Java virtual machine embedded within a dedicated microprocessor environment. Nevertheless, such systems are able to provide soft-real-time response.

Theoretical Foundations. The design of Real-Time Java has been based on the following existing real-time technologies:

1. Analysis of conservative worst-case execution times is straightforward for code segments which exclude recursion and unbounded loops [Nil95b]. However, on modern high-performance processors, it is very difficult to calculate an accurate worst-case execution time. Furthermore, the typical execution time of most code is much shorter than the worst-case execution time.
2. Measurement of representative function invocations in the local execution environment provides a more accurate representation of actual execution time on modern processors than pre-run-time analysis [Ken90]. Application developers are the best qualified to select function parameterizations that represent typical and/or worst-case invocations.
3. Given task execution times and frequencies, rate monotonic analysis allows quick and straightforward determination of whether a particular combination of real-time workloads is capable of running reliably within real-time constraints [Liu73].
4. Static cyclic schedules offer the benefit of lower run-time overhead and fewer context switches than fully dynamic scheduling [Xu90]. As a possible run-time optimization, static cyclic dispatch tables may be created by the real-time executive. Rate-monotonic techniques offer a straightforward approach toward creation of static schedules.
5. Real-time garbage collection can be configured in terms of total live memory and desired allocation rates for the combined workloads [Nil95c], [Nil94]. As long as all real-time tasks remain within their dynamic memory allocation budgets, each task can be assured that all allocation requests will be satisfied in a timely manner. The dynamic memory manager enforces a memory allocation budget for each real-time activity to prevent rogue tasks from interfering with the memory needs of other tasks.

Structure of a Real-Time Java Application. The standard distribution package for a real-time Java application is identified as a real-time activity. Conceptually, a real-time activity represents a complete service. Examples of real-time activities include text-to-speech synthesis, a video conference session, a video game, or a stock-market ticker. Note that each real-time activity may consist of several real-time tasks. For example, a video conference session might have five independent tasks to (1) record the local user's audio, (2) sample the local video camera, (3) display the incoming video stream, (4) playback the incoming audio stream, and (5) process all of the network packets associated with the video conferencing session. Each real-time activity consists of several real-time tasks, a configuration manager, and a resource negotiator.

Loading of a Real-Time Activity. When a real-time activity is loaded onto a Java virtual machine, the byte-code analyzer recognizes the code as representing a real-time activity and performs special analysis of the code. In particular, it builds a call graph for each of the real-time tasks that comprise the activity. For each of the methods (functions) that might be executed by the task, the byte-code analyzer examines all of the method's exception handlers and finally clauses to determine their worst-case execution times. The purpose of this analysis is to enable the real-time executive to deliver precisely timed abort requests to any task that might exceed its budgeted time slot. When a task is aborted, its pending exception handlers are allowed to execute. This gives the task an opportunity to release any mutual exclusion locks that it might hold and to perform other cleanup operations as necessary.

Configuration. After a real-time activity has been loaded and its byte codes analyzed, the activity can be added to the current workload. When a real-time activity is introduced onto a new host, the real-time executive for this host asks the activity to configure itself. Configuration consists of measuring the activity's resource needs in the local environment. During configuration, the activity is able to suggest that certain code segments be left in byte-code representation to conserve space and/or

reduce optimization efforts, and can suggest that other code segments be translated to native machine language so that they will run faster. Then, the activity's configuration manager measures the activity's resource needs. The configuration manager may invoke a built-in analysis tool to determine the worst-case execution time for blocks of code that do not contain unbounded loops, recursive method calls, or dynamically bound method invocations. Note that the analyzer is only able to determine worst-case execution times for a relatively small subset of the complete language. This is consistent with our assumption that measurement of representative task invocations provides much more accurate predictions of task execution times than analytical techniques in modern processing environments. For this reason, we discourage programmers from relying on analysis for all but relatively simple and short code segments.

Besides being able to request that the system determine worst-case execution time of selected segments through analysis, the configuration manager is also able to measure the time required to exercise arbitrary code sequences. Further, the configuration manager is able to enquire of the local system to determine the amount of memory required to represent particular data structures on this host.

In summary, the configuration manager is responsible for determining (1) the worst-case time required to execute particular segments of code written in a restrictive subset of the full language, (2) the typical time required to execute particular code segments (which may themselves represent typical or worst-case situations), and (3) the amount of memory required to represent particular data structures in the local environment. The configuration manager generalizes these values to determine the resource needs of this activity. It returns to the real-time executive a representation of the activity's resource needs in terms of the following quantities:

1. The minimum total amount of live memory that is necessary to support the activity.
2. The desired total amount of live memory that is necessary to support the activity.
3. The minimum acceptable rate for allocation of new memory (which is the rate at which the real-time garbage collector must be able to recycle dead objects).
4. The desired rate for allocation of new memory.
5. For each of the real-time tasks that comprise this real-time activity:
   a. The period of execution.
   b. The minimum amount of execution time required by this task in each period.
   c. The desired amount of execution time requested by this task in each period.
   d. The smallest acceptable time slice size (the total CPU-time allocation for each period of execution may be divided into multiple time slices no smaller than this quantity).

The real-time executive uses this information to determine whether there are sufficient resources to accept the workload represented by this new activity.

Resource Negotiation. Given representations of the resource requirements of each real-time activity in the system, the real-time executive computes CPU-time and memory budgets for each activity. The "standard" for Real-Time Java does not specify algorithms for analysis of schedulability, analysis of memory availability, for scheduling, or for garbage collection. We note, however, that today's technologies are sufficient to enable this process. In particular, real-time garbage collection algorithms are capable of guaranteeing to recycle memory at particular rates [Nil95], [Nil95c], and rate-monotonic analysis is capable of determining whether the combined workload, including the real-time garbage collection task, can be scheduled [Liu73]. Having analyzed the combined workload, the real-time executive offers time and memory budgets to each real-time activity. For each budgeted resource, the real-time executive grants a resource in terms of guaranteed and expected resource availability. There are many situations in which the expected resource availability

is much higher than can be guaranteed, and it is necessary for cost-effective high-performance application software to make efficient use of whatever resources are available even when they cannot be guaranteed.

Since the budgets offered by the real-time executive typically provide resource allocations greater than each activity's minimum acceptable requirement but less than the activity's requested ideal, the activity has an opportunity to accept or reject the proposed budget. If any activity rejects its proposed budget, the real-time executive may endeavor to propose an alternative more acceptable budget at its discretion. If, however, the real-time executive decides that it is unable to accommodate the new activity, it simply reports to the application that attempted to introduce this new activity into the environment that it was unsuccessful in its attempt. This application then has the option of removing other real-time activities from the system, requesting that certain activities be given preferential treatment during resource negotiation, and resubmitting its request to add the activity to the current workload. Resource negotiation itself is not a real-time activity. While resource negotiation is taking place, the system continues to run the previously negotiated real-time workload.

Execution of Real-Time Tasks. Each real-time task consists of essential startup and finalization code segments and a variable-duration work component. Both the startup and finalization code segments must be execution-time analyzable, and the task's minimum periodic execution time allotment includes at least enough time to execute these two segments. The work component's CPU-time request is typically determined by measurement, and its execution time budget typically consists of a combination of guaranteed and unguaranteed execution time.

The work component may include arbitrary Java code, including unbounded loops and unbounded recursion. It is important to make this allowance because developers of real-time software components often desire to make use of general purpose reusable object libraries designed for high performance and high-level abstraction. Use of these libraries reduces software development time and improves system flexibility, but automated analysis of worst-case resource requirements is impractical. It is the responsibility of each real-time activity's resource configuration and negotiation components to arrange for the amount of resources required to achieve whatever level of service quality is considered practical.

Within each period, the real-time executive allows the work component of each task to run for as much of the task's requested ideal execution time as is available. The real-time executive aborts the task when the available time is exhausted, always leaving sufficient time to execute the work component's pending exception handlers followed by the task's finalization code.

Blocking. It is well known that schedulability analysis is greatly complicated by analysis of intertask blocking. To cope with the highly dynamic nature that we expect to be typical of Real-Time Java environments, we have adopted a mutual exclusion model in which the partitioning of CPU time is sufficient to enforce most of the mutual exclusion needs of real-time tasks. The standard mutual exclusion mechanism is an atomic statement. While the body of one atomic statement is executing, no other atomic statements belonging to the same object are allowed to execute. We require that the body of each atomic statement be execution time analyzable. On a single processor Java virtual machine, the implementation of atomic statements is straightforward. If there is sufficient time in the current time slot to execute the atomic statement, we do so. Otherwise, we suspend the task until its next scheduled time slot. With multiprocessor implementations, additional run-time overhead may be required to prevent interleaved execution of atomic statements. But the general philosophy is the same: No task is allowed to hold a lock during times when the task is not executing. This protocol eliminates priority inversion problems, and makes analysis of blocking an intratask concern.

REFERENCES

[Alf85] Alford, M., *SREM at the Age of Eight*, IEEE Computer, April 1985, pp. 36 - 46.

[Aig95] Aigner, G., Holzle, U., *Eliminating Virtual Function Calls in C++ Programs*. 1995. Department of Computer Science, University of Santa Barbara: Santa Barbara, CA.

[Bri90] Britton, D.E., *CASCADE a Dynamic Visual Environment for the Design of Computers*, Master Thesis, George Mason University, 1990.

[Bro96] Brockmeyer, M., Jahanian, F., Heitmeyer, C., Labaw, B., *An Approach to Modeling and Assertion-Checking of Real-Time Specifications* in Proceedings of the 4th International Workshop on Parallel and Distributed Real-Time Systems, April 15 -16, 1996, Honolulu, Hawaii

[Buh95] Buhr, R.J.A., Casselman, R.S. Pearce, T.W., *Design Patterns with Use Case Maps: A Case Study in Reengineering an Object-Oriented Framework*, SCE 95-17, http://ftp.sce.carleton.ca/UseCaseMaps/dpwucm.ps, 1995

[Buh96] Buhr, R.J.A., *Use Case Maps for Attributing Behavior to System Architecture* in *Proceedings of the 4th International Workshop on Parallel and Distributed Real-Time Systems*, April 15 -16, 1996, Honolulu, Hawaii

[Buh96a] Buhr, R.J.A., Casselman, R.S., *Use Case Maps for Object-Oriented Systems*, Prentice Hall, 1996

[Cas93] *SRS for the Computer Aided System and Component Architecture Design Environment*, Version 1.1., Trident Systems Inc, 1993.

[Dem78] DeMarco, T., *Structured Analysis and System Specification*,1978, Yourdon Inc., New York, NY.

[Det94] Detlefs, D., Dosser, A., Zorn, B., *Memory Allocation Costs in Large C and C++ Programs*. Software Practice and Experience, 1994. 24(6):p. 527-542.

[Har87] Harel, D., *Statecharts: a visual formalizm for complex systems*, Science of Computer Programming, 1987, vol 8.

[Hat88] Hatley, D.J., Pirbhai, I.A., *Strategies for Real-Time System Specification*, 1988, Dorset House

[Hub86] Huber, P., Jensen, A.M., Jepsen, L.O., Jensen, K., *Reachability Trees for High-level Petri Nets*, Theoretical Computer Science 45, 1986, pp. 261-292.

[Hoa85] Hoare, C.A.R., *Communicating Sequential Processes*, 1985, Prentice Hall

[Hoa91] Hoang, N., *The Essential Views of the System*, System Design Synthesis Technology Workshop, 1991, Silver Spring, MD.

[Isa96] Isazadeh, A., Lamb, D.A., MacEwen, G.H., *Viewcharts: A Behavioral Specification Language for Complex Systems* in *Proceedings of the 4th International Workshop on Parallel and Distributed Real-Time Systems*, April 15 -16, 1996, Honolulu, Hawaii

[ISO92] ISO. *Basic Reference Model of Open Distributed Processing - Part 2: Descriptive Model*, December 1992.

[Jac92] Jacobson, I. et al., *Object-Oriented Software Engineering (A Use Case Driven Approach)*, ACM Press, Addison-Wesley, 1992

[Jah88] Jahanian, F., Lee, R.S., Mok, A.K., *Semantics of Modechart in Real-Time Logic,* in *Proceedings on 21st Hawaii International Conference on System Science*, 1988.

[Jah94] Jahanian, F., Mok, A.K., *Modechart:A specification language for real-time systems,* IEEE Trans. Software Engineering, vol 20, no. 10, 1994

[Kar92] Karangelen, N.E., Hoang, N.D.T., *Sample Sonar System Example Problem Capture*, 1992 SSSWG Meeting Review Copy, Silver Spring, MD.

[Kar94] Karangelen, N.E., Hoang, N.D.T., *Representing System Behavior in Design and Analysis of Large Complex Real-Time Systems* in *Proceedings - Second Workshop on Parallel and Distributed Real-Time Systems*, April 28 -29, 1994, Cancun, Mexico

[Ken90] Kenny, K.B., Lin, K.J., *A measurement-based performance analyzer for real-time programs*, 1990, University of Illinois at Urbana-Champaign: Urbana, IL.

[Li94] Li, G., Bacon, J., *Supporting Distributed Real-Time Objects* in *Proceedings - Second Workshop on Parallel and Distributed Real-Time Systems*, April 28 -29, 1994, Cancun, Mexico

[Lin90] Lindquist, M., *Parametrized Reachibility Trees for Predicate/Transition Nets*, in *11th International Conference on Applications and Theory of Petri Nets*, 1990, Paris

[Liu73] Liu, C.L., Layland, J. W., *Scheduling Algorithms for Multiprogramming in a Hard-Real-ime Environment*, Journal of the ACM, 1973. 20(1): p.44-61.

[Nil94] Nilsen, K., *Reliable Real-Time Garbage Collection of C++.,* Computing Systems, 1994. 7(4): p. 467-504.

[Nil95] Nilsen, K. *Progress in Hardware-Assisted Real-Time Garbage Collection.* in *Lectures on Computer Science*. 1995. Kinross, Scotland:Springer-Verlag

[Nil95a] Nilsen, K. *Real-Time is No Longer a Small Specialized Niche.* in *Fifth Workshop on Hot Topics in Operating Systems (HotOS-V).* 1995. Orcas Island, Washington: IEEE Computer Society Press.

[Nil95b] Nilsen, K., Rygg, B. *Worst-Case Execution Time Analysis on Modern Processors*. in *ACM SIGPLAN 1995 Workshop on Languages, Compilers, and Tools for Real-Time Systems*. 1995. San Diego, California: ACM SIGPLAN

[Nil95c] Nilsen, K., Gao, H., *The Real-Time Behavior of Dynamic Memory Management in C++, in IEEE Real-Time Technologies and Applications Symposium*. 1995. Chicago, IL.

[Nil96] Nilsen, K., *Issues in the Design and Implementation of Real-Time Java*, July 19, 1996. http://www.newmonics.com/WebRoot/technologies/java/RTJI.ps

[Nil97] Nilsen, K., *Embedded Real-Time Development in the Java Language*, http://redwood.snu.ac.kr/PAPERS/rtjava/EmJava.ps

[Ran94] Randell, B., Ringland, G., Wulf, B., *Software 2000: A View of the Future*. 1994 Design to Distribution Ltd.:Stevenage, Herys, England

[Reg96] Regnell, B., Andersson, M., Bergstrand, J., *A Hierarchical Use Case Model with Graphical Representation* in *Proc. ESBC96, IEEE Second International Symposium and Workshop on Engineering of Computer Base Systems*, March 1996.

[Rei82] Reisig, W., *Petri Nets An Introduction*, EATCS, Monographs on Theoretical Computer Science, 1982, Springer-Verlag

[Sel94] Selic, B., Gullekson, G., Ward, P.T., *Real-Time Object-Oriented Modeling*, 1994, Wiley

[Sel96] Selic, B. *Modeling Real-Time Distributed Systems* in *Proceedings of the 4th International Workshop on Parallel and Distributed Real-Time Systems*, April 15 -16, 1996, Honolulu, Hawaii

[Sch94] Schmidt, W.J., Nilsen, K. *Performance of a Hardware-Assisted Real-Time Garbage Collector*, in *Sixth International Conference on Architectural Support for Programming Languages and Operating Systems*. 1994. San Jose, CA.

[Spi92] Spivey, J.M., *The Z Notation - A Reference Manual*, Second Edition, !992, Prentice Hall

[Xu90] Xu, J., Parnas, D. L., *Scheduling Processes with Release Times, Deadlines, Precedence, and Exclusion Relations*. IEEE Transactions on Software Engineering, 1990. 16(3): p. 360-369.