AN ARCHITECTURAL FRAMEWORK FOR DISCRETE EVENT SIMULATION USING KNOWLEDGE-BASED REASONING

by

Lisa Winkler

An Applied Project Presented in Partial Fulfillment

of the Requirements for the Degree

Master of Computer Science

ARIZONA STATE UNIVERSITY

May 2005

REPORT OF FINAL MASTER'S WRITTEN OR ORAL EXAMINATION
ARIZONA STATE UNIVERSITY
DIVISION OF GRADUATE STUDIES

Instructions.

1. Part I. The student completes Part I and submits the Report to the Academic Unit following all academic unit deadlines and procedures. The student's Program of Study must be approved by the Division of Graduate Studies before the examinations are scheduled.
2. Part II. After the examination, the examining committee members complete Part II, providing the date of the examination, signing the form, and indicating their votes of Passed or Failed.
3. Part III. The head of the academic unit completes Part III noting the final result of the examination(s). The final result is based on the majority vote of the committee members.
4. Notification. The academic unit sends the student a written statement of the results of the examination.
5. Submission. The completed Report should be submitted immediately to the Division of Graduate Studies, Wilson Hall Lobby, mail code 1003.

   All test results, including failures, should be reported. Failure in an examination is considered final unless the student petitions for a re-examination, the supervisory committee and the head of the academic unit recommend, and the Division of Graduate Studies Dean approves a retake.

PART I. Student Information. To be completed by the student. Please print or type.

NAME OF STUDENT (Last name, first name, middle initial)	ASU I.D. NO.
Winkler, Lisa M	164-60-3437
ADDRESS (NO., STREET, APT.)	TELEPHONE
4102 E Juanita Ave	480-857-7068
CITY, STATE, ZIP	
Gilbert, AZ 85236	
MASTER OF	MAJOR
Computer Science	
SIGNATURE Lisa M Winkler	DATE 5-6-05

Part II. Examination Date and Result. To be completed by the examining committee.

TYPE OF EXAMINATION		
☐ ORAL ☑ WRITTEN		
DATE OF EXAMINATION	TIME	PLACE
5-6-05		

PRINT/TYPE NAMES OF EXAMINING COMMITTEE	SIGNATURES	PASSED (✓)	FAILED (✓)
CHAIR OR CO-CHAIR Hessam Sarjoughian		☑	☐
CO-CHAIR Huan Liu		☑	☐
MEMBER		☐	☐
MEMBER		☐	☐
MEMBER		☐	☐
MEMBER		☐	☐
MEMBER		☐	☐

Part III. Final Result.

PASSED	FAILED	SIGNATURE, HEAD OF ACADEMIC UNIT	DATE
☑	☐		7/15/05

ABSTRACT

Software architects use architectural styles to help them select an appropriate design for a particular problem domain. Shaw & Clements have developed a classification system that allows the architect to choose from a number of common styles based on certain aspects of the problem domain, or develop their own based on a specific set of architectural attributes. Knowledge of these architectural attributes, in turn, allows the architect to make some inferences and decisions about the quality attributes of the resulting system.

In this paper we apply these software architecture principles to perform an analysis of a class of systems composed of a simulation environment and a knowledge base. The integration of simulation and knowledge-based systems poses interesting architectural challenges since they have some significant differences in their architectural attributes. In this report we examine Discrete Event System Specification (DEVS) simulation and Reactive Action Planner (RAP) agent in comparing candidate architectural styles. We will take a detailed look at the architectural attributes of each component, and use them to select a few candidate styles for a combined system called DEVS/RAP. We will then compare the quality attributes of each of these design options.

TABLE OF CONTENTS

LIST OF TABLES

1 Introduction

1.1 Software Architecture

Software architecture refers to the components into which a system is divided at a gross level of system organization, and the ways in which those components behave, communicate, interact, and coordinate with each other [3]. The software architecture represents the highest level of design decisions about a system. An architectural style refers to a set of design rules that identify the kinds of components and connectors that may be used to compose a system or subsystem, together with local or global constraints on the way the composition is done [13].

1.2 Architectural Attributes

Shaw and Clements have developed a classification system for architectural styles [13]. This classification system begins by identifying components and connectors, as the primary discriminators among styles. A component is a unit of software that performs some function at run-time, such as an object or process. A connector is a mechanism that mediates communication, coordination, or cooperation among components, such as a procedure call or messaging protocol. The classification system then moves on to examine control, data flow, and control/data interaction attributes. The combination of these attributes comprises the distinguishing features of an architectural style, and allows for rigorous comparison among architectural styles. This analysis provides the architect with the ability to identify significant differences that affect the suitability of styles for various tasks, and assists in selection of an appropriate style for a given problem.

1.3 Quality Attributes

In order to determine the suitability of an architectural style for a particular problem, it is important to identify the quality attributes that are of high priority with regard to the particular problem domain, and compare possible solutions in terms of these attributes. A large number of quality attributes have been studied. For the purposes of this paper, we will be considering only a select few attributes that are deemed relevant to the problem at hand. (A rationale for the selection of these particular attributes will appear later in the paper.)

1.3.1 Performance

Performance is the attribute of a computer system that characterizes the timeliness of the service delivered by the system [1]. When working in a simulation environment, performance is of great interest because it affects the ability of the system to adequately represent real-world scenarios. A simulator would not be of much use if it were unable to function in a timely manner as compared to the system it is trying to simulate. And, the interface between the simulator and knowledge base must perform at a sufficient level to allow the simulator to query the knowledge base, receive a response, and act upon it in a timely manner.

The performance of the system may be greatly affected by the frequency of messages that need to be exchanged between components of the system, especially if the components are physically distributed processes [3]. The messaging frequency may vary widely due to system design/implementation decisions, varied usage scenarios, or loading on the system. It is important to consider the message frequency aspects of a variety of system implementations and scenarios when evaluating the performance attributes of an architectural solution.

1.3.2 Scalability

Scalability is the ability of a system to support modifications that dramatically increase the size of the system [2]. In terms of the simulation/knowledge base environment, it may be useful to be able to support multiple associations between different simulations or knowledge bases. Scalability is also related to performance in the sense that an architectural solution may have some limitations when the size of the system and/or the number of components increases such that the frequency of messages between components exceeds the capabilities of the architecture.

1.3.3 Modifiability

Modifiability depends extensively on the system's modularization, which reflects the encapsulation strategies [3]. An architectural style that provides for strict encapsulation between components is likely to be more modifiable than one that defines loosely defined boundaries or interfaces between components. Encapsulation is not just an object-oriented design principle, then; it is a means of enabling modifiability, which contributes to the quality of an architectural solution.

2 Background

The primary focus of this paper will be to analyze architectural solutions within a specific problem domain, namely that of combining a discrete event simulator with a knowledge base agent execution system [11][12]. To help ground the discussion, we restrict it to a specific implementation of each of these types of systems.

2.1 DEVS

The Discrete EVent System Specification (DEVS) is a modeling formalism that describes a mathematical model for a real-world system [14][15].

A DEVS model of a particular system is composed of atomic models and coupled models. Each atomic or coupled model represents a system component. An atomic model consists of a set of transition functions, which describe the behavior of the model, and a set of ports which allow for input and output between this model and other models in the system. Communication between atomic models is accomplished by passing messages – a message produced on a particular output port of one model is consumed on the input port of another model. Each atomic model also has one or more states, including a special state called sigma, which indicates the time remaining until the next scheduled state transition. The timing of the model is represented by the time advance function. Atomic models may be grouped into coupled models, which have their own sets of input and output ports, and may be connected via these ports to other atomic or coupled models.

Execution of a DEVS model requires a discrete event simulator such as DEVSJAVA [5]. The simulator provides the algorithm and environment for executing the model, by accepting inputs from a user or external system, routing them to the correct models based on the specified connections, and similarly directing outputs of one model to the appropriate inputs of another model or to an external system. The simulator also keeps track of the time elapsed throughout the simulation, and ensures that time is synchronized among all models involved in the simulation.

DEVS and DEVSJAVA are described in further detail in [14]. For the purposes of this paper, it is sufficient to understand that a DEVS model consists of components which are atomic or coupled models, and connectors which are ports and messages. This results in a hierarchical composition of components.

Hereafter, the term DEVS will be used to refer to the DEVSJAVA simulator rather than the DEVS formalism itself. It is understood that the DEVSJAVA simulator is configured to execute a system of atomic or coupled models.

2.2 RAP

A Reactive Action Package is a representation that groups together and describes all of the known ways to carry out a task in different situations [6]. The Reactive Execution Planner, or RAP, is a system that manages a collection of Reactive Action Packages. It was originally designed to function as the reasoning engine of a software system that controls a robot. The RAP system accepts input from the robot's sensors, and the its components include the memory, the interpreter, and the RAP library. The RAP library is simply the collection of Reactive Action Packages available to the system. The RAP memory contains the execution system's best assessment of the current situation. The RAP interpreter coordinates task execution, selecting active tasks or packages from the library depending on the current situation and the goal to be accomplished.

The RAP system is implemented in C++, but also allows interaction from an external system or user by using the RAP language [7]. This is a specialized, logic-oriented language used for defining Reactive Action Packages, manipulating the RAP task list, and querying the RAP memory and/or task execution status. An external system can thus query the RAP system for information it has already discovered about the environment, or direct it to perform tasks that result in new information or changes to the environment.

2.3 DEVS/RAP

DEVS is useful for modeling event-driven systems, but its behavior is controlled mainly by its external and internal transition functions which are somewhat limited in the sense that the only information available to these functions comes from the input messages it receives [11]. If the real-world environment to be modeled is changing throughout the simulation, it is difficult to account for such dynamic behavior at specification time without intelligent algorithms; another area of research entirely.

RAP is a powerful tool for reasoning about an environment, but it is primarily designed for interacting with a dynamic, real-world environment, receiving input through sensors. It would

sometimes be useful to allow RAP to interact with a simulation environment, for example, for the purpose of testing either the simulation or the knowledge base.

Consider a simple example. A car is traveling in a major city containing a complex network of roadways. The environment is quite dynamic since other vehicles and people are acting as obstacles, roads may be closed due to construction, etc. A simulation environment that understands the basics of operating a car (applying gas/brake, steering, etc) could be constructed fairly easily using DEVS and its transition functions. However, including the knowledge about how to navigate from one place to another would be very challenging to do in DEVS due to the complexity and dynamics of the environment. Also, if the knowledge base about the environment were embedded within DEVS, it would make it very difficult to reuse the DEVS simulator within another environment (i.e., a different city). It would be useful, then, to be able to abstract the knowledge and reasoning needed for navigation from the mechanics of operating the vehicle using a system designed expressly for this purpose.

In this paper, we will investigate architectural styles for composing a simulation environment (DEVS) with a knowledge base (RAP) given an implementation of DEVS/RAP which combines DEVSJAVA simulation environment with RAP execution system [11].

3 Architectural Considerations for DEVS/RAP

Much of the work on architectural analysis of a problem domain begins with identifying components and connectors. For the DEVS/RAP system [11][12] considered in this paper, this step is fairly trivial. The components are, of course, DEVS and RAP. The most interesting aspect, then, is the choice of connectors for these components.

3.1 Key Architectural Attributes

In order to propose an architectural style for a system involving RAP and DEVS, it is helpful to first understand some of the attributes of each of these components. These attributes will influence the choice of connectors for the components.

		DEVS	RAP
Constituent Parts	**Components**	atomic/coupled models	tasks, packages
	Connectors	messages, ports	procedure calls
Control Issues	**Topology**	hierarchical	arbitrary
	Synchronicity	asynchronous	asynchronous
	Binding Time	write	run
Data Issues	**Topology**	hierarchical	arbitrary
	Continuity	sporadic	sporadic
	Mode	passed	shared
	Binding Time	write	run
Control/Data Issues	**Isomorphic Shapes**	yes	no
	Flow Directions	same	NA
Types of Reasoning		Procedural	declarative
Timing		controlled	uncontrolled

Table 3-1 - Architectural Attributes of RAP & DEVS

This section looks at three attributes in detail that significantly constrain the architecture of this system.

3.1.1 Connectors

In this context, the connectors of interest are the connectors of RAP and DEVS that form the external interfaces of these components. It is these external interfaces that will determine the types of connectors that may be used in the combined system, because they constrain the control flow and data flow between the components.

The DEVS connectors consist of ports and messages that may be addressed to a particular port. The atomic or coupled models within DEVS may be connected to external input and output ports to form an interface between the DEVS environment and external systems.

As a logic-based system, RAP is procedurally oriented. An external system interfacing with RAP will execute queries or commands to manage RAP's task network.

The dissimilarity between the external interfaces comprises one of the most significant challenges in integrating these two components. One basic tenet of software architecture and design is to avoid violating the encapsulation of existing components. So, it will be necessary to provide some type of connector that can adapt to both interfaces and translate between them.

Message based connectors can degenerate into procedure calls, but it's hard to do the reverse. So, the resulting system must use procedure calls as its interface into RAP. It is possible to wrap RAP in a messaging interface, but the wrapper would still have to use procedure calls to invoke RAP behavior.

3.1.2 Synchronicity

Both RAP and DEVS are asynchronous in nature, so theoretically they could each behave independently. However, in a DEVS/RAP system, they are really complementary models of the same environment. In this environment, DEVS represents the "Physical model", capturing the behavior of the environment, while RAP represents the "Agent model", which has the ability to reason and make decisions about its environment. Returning to the car example, the physical model involves real, physical operations and constraints such as steering a vehicle. The agent model is concerned with navigating the vehicle from point A to point B, identifying and avoiding obstacles, etc. Because these models are both tied to the same real-world scenario, it is clear that the two components cannot be allowed to behave independently. Their behavior must be synchronized with each other or the semantics of the system will break down.

3.1.3 Timing

This architectural attribute is an extension of the list developed by Shaw and Clements. For many architectural styles, the synchronicity attribute is sufficient to classify the timing aspects of the architecture. However, when considering a simulation environment, which may be closely related to real-world behavior, timing becomes much more critical.

The following classification for timing is proposed:

- Controlled: The component controls time internally. DEVS can arbitrarily pause or advance time during execution of a simulation; as a result, the passage of simulation time may have little correlation to time outside the simulation environment.

- Uncontrolled: The component allows time to be controlled by its external environment. For example, RAP tasks may take an arbitrary amount of time to execute. Although RAP does support the ability to measure or bound the length of time spent on a particular task, the time taken is simply measured against the system clock.

Clearly, if timing is critical to the problem domain, then the timing attribute of all components must be compatible. If only one component controls its own timing while the rest have uncontrolled (externally controlled) timing, it may be feasible to resolve this incompatibility by allowing the former to control the timing for the entire system.

3.2 Other Attributes

The preceding section identified several attributes of RAP and DEVS that are most significant in the choice of connectors. The remaining attributes have little influence on the selection of connectors, largely because these attributes do not significantly affect the interfaces between components. Because DEVS and RAP each have well-defined external interfaces, a system comprised of these two components need not be greatly concerned with attributes that primarily describe their internal structure/behavior. However, for a complete architectural analysis it is necessary to briefly address these attributes.

3.2.1 Control/Data Topology

Because DEVS is comprised of atomic models which may be composed into coupled models at an arbitrary depth, it is classified as having a hierarchical topology. Since the flow of control and data are isomorphic, this argument can apply to both the control and data topology. By contrast, RAP consists of tasks and assertions which may be related to each other in arbitrary ways; therefore, it is classified as having an arbitrary topology. Furthermore, the flow of control is arbitrary in the sense that task ordering is generally not known at specification time and in fact can change dynamically during execution in response to external events. The internal topology can be safely ignored with the assumption that each component will return control and/or data

flow to the other within a finite amount of time. It is certainly possible to construct a DEVS model and or RAP task network such that this assumption is invalid; however, for purposes of simplifying the discussion we will assume that such a construction will not be used in a DEVS/RAP system, as this would not be a particularly meaningful or illustrative example.

3.2.2 Control Binding Time

In DEVS, control flow is bound at compile time, since couplings between models must be defined at compile time unless the model can have variable structure (e.g., components and couplings can be removed at run time [15]). For RAP, however, tasks and memory can be initialized prior to run time if desired, but may be changed at any time during execution. If some external agent were to modify RAP's memory, this could cause inconsistencies between RAP and DEVS. Therefore, for simplification purposes we will assume that DEVS is the only agent that can modify RAP memory. This assumption allows us to safely rule out control binding time as a significant factor in selecting a DEVS/RAP architecture.

3.2.3 Data Continuity

For both components, data may enter and move through the system sporadically. DEVS may receive external messages at a rate that is limited only by the simulation software; RAP may receive input from the real world or simulation environment as it becomes available. Therefore the components are compatible in terms of data continuity.

3.2.4 Data Mode

In DEVS, data is passed using messages; in RAP it is shared using a memory pool that is accessible to all tasks. However, the RAP language does provide an interface that allows an external system to manipulate RAP memory in terms of assertions, which turns out to be quite compatible with DEVS messaging interface. Therefore, the dissimilarity in internal data mode will not have a significant impact on the choice of connectors.

3.2.5 Isomorphic Shapes

As noted in section 3.2.1, DEVS control/data flows are isomorphic because control and data pass from component to component essentially at the same time. Since RAP's control and data topology are arbitrary, so must be the control/data flow – and therefore these cannot be assumed to be isomorphic.

3.2.6 Flow Directions

Since the control and data flow of both components are arbitrary, RAP and DEVS can assume to be compatible in this respect based on the same assumptions as in section 3.2.1.

3.2.7 Types of Reasoning

DEVS reasoning can be thought of as procedural, since its execution is driven by a set of functions that define each atomic model. RAP reasoning, by contrast, is declarative – RAP tasks execute by following a set of rules. Again, this attribute can be ruled out as a contributing factor based on the same constraints discussed previously; the type of reasoning used internally to the component is not critical as long as the interface between them is well-defined.

3.3 Other Constraints

This section looks at some other constraints and considerations that are not directly tied to architectural attributes, but nonetheless will affect the selection of an architectural style.

3.3.1 Ordering

For message-based connectors, ordering is an important consideration. The ordering of messages received on an interface may affect how those messages are processed by the receiving component. In DEVS, ordering is not important, messages are passed around in a "bag". However, in RAP, ordering of messages must be enforced because processing of incoming messages may result in changes to RAP memory. Therefore, it is important that any architectural solution ensure that ordering of messages delivered on the RAP interface is maintained.

3.3.2 Separation

Separation is a basic architectural concept that involves placing a distinct piece of functionality into a distinct component that has a well-defined interface to the rest of the world [10]. To maintain separation, any architectural solution must comply with the interfaces of each component, and should not require internal changes to these components.

Another reason for concern about separation and maintaining the encapsulation of DEVS and RAP is that both of these components are implemented using well-studied and tested algorithms. Any internal changes to these components could undermine the consistency of these algorithms and render the resulting system less reliable than its constituent parts.

3.3.3 Language/Technology

Since the two components to be integrated have been developed in different languages (C++ and Java), the choice of mechanism for integrating them may be somewhat restricted by the available technology. Some possible integration technologies include JNI [9], middleware (such as CORBA [4] or J2EE [8]), or portable libraries. While the selection of integration technology can and should be left as a design decision, it may be helpful to understand the basics of some of these technologies when developing the architecture, since this can help to ensure that the architecture is implementable.

4 Architectural Options

In this section we consider several possible architectural choices for a DEVS/RAP system. Table 4-1 summarizes each of these architectural styles in terms of its attributes, highlighting those key attributes identified in the previous section. The following subsections describe each proposed option and summarize its key attributes.

		Tightly Coupled	Distributed	Bridge
Constituent Parts	**Components**	DEVS, RAP	DEVS, RAP	DEVS, RAP
	Connectors	procedure calls	messages	messages
Control Issues	**Topology**	arbitrary	arbitrary	cyclic
	Synchronicity	synchronous	asynchronous	synchronous
	Binding Time	write	run	write
Data Issues	**Topology**	arbitrary	arbitrary	cyclic
	Continuity	sporadic	sporadic	sporadic
	Mode	passed	passed	passed
	Binding Time	write	run	write
Control/Data Issues	**Isomorphic Shapes**	yes	yes	yes
	Flow Directions	same	same	same
Types of Reasoning		procedural	procedural	procedural
Timing		controlled	uncontrolled	controlled

Table 4-1 - Attributes of DEVS/RAP Architectural Options

4.1 Tightly Coupled

In a tightly coupled architecture, the relationship between the components (DEVS/RAP) will be fixed at specification time. Due to the complexity of the simulation cycle, it is important that DEVS maintain control of the timing and messaging, interacting with RAP as necessary. DEVS will communicate with RAP using procedure calls to manipulate RAP tasks and assert facts from RAP memory. Such calls would occur within the DEVS simulation cycle. While it may already be fairly obvious to the reader that this solution will result in some significant limitations, it is illustrative to compare it to other, more flexible solutions.

Connectors: Procedure calls
Synchronicity: Synchronous (since procedure calls are synchronous by nature)
Timing: Controlled (DEVS controls timing)

4.2 Distributed

A distributed architecture for this system would, by definition, support RAP and DEVS components that are decoupled physically as well as logically. Components may reside on

physically separate machines connected by a network. It may even be possible for one DEVS to choose among many possible RAPs at runtime, or vice versa. Such an architecture would require some type of middleware to facilitate discovery of and communication between components. As is the case with most modern distributed systems, this architecture would use messages to pass data between components. A wrapper would be used to translate these messages into procedure calls on behalf of RAP, and to translate RAP output into messages that may be passed to DEVS.

However, a distributed solution has one major obstacle: timing. As noted earlier, control of timing is critical to DEVS execution; however, in an asynchronous environment, it is not possible to guarantee control of timing. Therefore, the timing attribute must be *uncontrolled*. A discussion of the implications of this issue is beyond the scope of this paper; future work would be needed to assess the feasibility of this option. At present, we can discuss the relative merits of a distributed solution as justification for future work in this area.

Connectors:	Messages (facilitated by middleware)
Synchronicity:	Asynchronous (distributed architectures using messaging are typically asynchronous in nature unless additional constraints are placed to enforce synchronicity; this is not proposed)
Timing:	Uncontrolled

4.3 Bridge

A bridge architecture provides somewhat of a compromise between the preceding two options [11][12]. The bridge would provide a connection between DEVS and RAP, specified at compile time. The bridge (also referred to as Knowledge Interchange Broker (KIB)) offers a generic architecture for discrete event (DEVS) and agent (RAP) models to exchange data and control information given their respective formalisms. To deal with the timing issues, this architecture takes a somewhat simplified approach by requiring a synchronous interface between RAP and DEVS. As noted earlier, since message-based connectors can easily degenerate into procedure calls by using a wrapper/translator mechanism, it is reasonable to take advantage of DEVS messaging interface and place the translation of these messages into RAP procedure calls, into the bridge.

Connectors:	Messages
Synchronicity:	Synchronous
Timing:	Controlled

4.3.1 Bridge implementation

The bridge approach, called the "Knowledge Information Broker", or KIB, has actually been implemented, so we can examine it in more detail [11]. Figure 4.3.1 illustrates the conceptual architecture.

1) DEVS outputs a message containing a RAP query or command
2) KIB calls appropriate RAP library function based on contents of message
3) RAP puts results on message queue for DEVS
4) KIB delivers message to DEVS

Figure 4.3.1- DEVS/RAP Bridge Architecture

The KIB allows DEVS to send messages to RAP through an output port that is connected to DEVS' external interface. The KIB translates these messages into RAP commands or assertions, and executes the resulting operation using RAP's C++ API. The result from RAP is then put into a DEVS message and placed on an external DEVS input port. This allows DEVS to receive the message and have it routed to an appropriate model for processing, just as it would receive an input from any other system. The result is a cyclical message flow from DEVS to RAP, and back to DEVS. The interaction with RAP, then, does not interfere with the normal simulation cycle that occurs as part of the DEVS execution. This is important because it ensures that the already-proven simulation algorithm will still work even though it is now part of a larger system.

When DEVS sends a message, the simulation is stopped and does not continue until the corresponding RAP operation has completed. This eliminates the need for keeping two separate clocks in synch; from DEVS' perspective, the RAP operation executed instantaneously. This simplification is reasonable because, since DEVS controls its own time, any time taken by RAP can be considered irrelevant to the simulation time.

The KIB enforces ordering among messages that are received from DEVS. This is trivially seen since the simulation cycle is stopped while RAP is processing a DEVS request; the simulator can not possibly generate any more messages during this time. As noted above, this ensures that any

messages that may involve changes to RAP's memory or task list will be interpreted in the correct order.

An additional convention that is added for simplification purposes is to allow DEVS to control the messaging cycle. DEVS always initiates a request for RAP and waits for a response, never the reverse. This convention greatly decreases the complexity of the implementation, but it also somewhat limits the set of problems to which this implementation can be applied. The simulation must be fairly autonomous, but may consult RAP periodically. The reverse scenario would be to allow RAP to operate autonomously, and request information from DEVS as needed. While this is certainly possible, and could likely be implemented with only minor changes to the architecture and design, it has not yet been developed.

5 Quality Attributes

Given the problem domain described earlier, we can identify quality attributes that are important to this domain, and qualitatively analyze each of the architectural options in terms of its architectural attributes and quality attributes. A qualitative analysis should be sufficient for choosing an appropriate architecture for a prototype implementation; a quantitative analysis is beyond the scope of this paper.

The set of quality attributes that follow were chosen for analysis based on the current research status of these components and the expected application of the system being proposed. Both RAP and DEVS are used primarily in the simulation and test domain. Therefore, quality attributes such as security, reliability, maintainability, etc which might be considered quite important to a production system are of only indirect interest here. Instead, we are primarily focused on the feasibility and practical applications of the DEVS/RAP system. Therefore, those quality attributes most likely to impact the usefulness of DEVS/RAP in enabling future research are most relevant to this discussion.

5.1 Performance

Kazman, et al [10] note that separation into components (such as this case in which the system is made up of existing components) may be a “plus” or “minus” for performance; this is one attribute that is likely to be determined by specific problem scenarios or implementation as well as architectural attributes. In this case, the performance is greatly affected by the frequency of interaction between RAP and DEVS, which depends largely upon the specific environment to be simulated. Further analysis of messaging scenarios for a specific problem domain would be needed before applying any of the proposed architectures to that domain. However, at a high level we can examine the architecture for bottlenecks in data flow or processing.

In the tightly-coupled approach, RAP will be invoked using a direct procedure call each time information is needed; this could introduce a significant amount of memory and/or processing overhead to construct and deconstruct any execution environment and/or infrastructure that is needed on each call. This overhead coupled with the synchronous nature of the procedure calls could greatly affect the amount of real-world time necessary to execute a simulation. The bridge itself is likely to create a similar bottleneck, especially if there is a high frequency of communication between the two primary components or large amounts of data being passed;

however, allowing the bridge to manage a single RAP instance and minimizing memory and processor usage will help to alleviate this issue. Because a distributed solution will allow DEVS and RAP to execute on separate hardware, this can have a positive impact on performance by preventing the two components from competing for resources. However, performance may also be negatively impacted by network latency and/or middleware overhead. Again, the tradeoffs of a distributed solution would need to be studied in the context of the problem domain and perhaps the selection of middleware for implementation.

5.2 Scalability

A procedure call, by definition, allows for a 1:1 mapping between components. This is clearly a significant limitation of the tightly coupled architectural style. In contrast, a messaging interface allows for an n-way mapping between components, greatly improving scalability. Both the bridge or distributed architecture utilize a messaging interface; however, the bridge architecture still supports only a 1:1 mapping due to timing assumptions as pointed out earlier. Therefore, the distributed architecture, supporting an n:n mapping between components, is clearly the most scalable in terms of its interface. Provided that the middleware is well-designed, it has the potential to support a large number of RAP-DEVS associations with minimal performance degradation. In addition, it allows for a heterogeneous mixture of simulation environments and knowledge bases.

Synchronicity also affects the scalability of an architectural solution. A synchronous solution is generally less scalable than an asynchronous one, since a synchronous solution lacks the flexibility to adjust its behavior based on the number or types of requests it receives. Again, the distributed solution is likely to be the most scalable since it allows for asynchronous execution. However, practical considerations make it extremely difficult to implement an asynchronous solution while maintaining the integrity of the simulation environment.

5.3 Modifiability

Since RAP and DEVS are each but one possible implementation of a simulation environment and knowledge base, it would be desirable when defining an architecture to allow for the possibility of plugging in new or varying implementations of each of these components with a minimal amount of effort. The choice of connectors has the greatest impact on modifiability of those architectural attributes identified. A messaging connector provides the most flexibility for future

modification since it provides a layer of abstraction between the data being passed and the producers and consumers of this data; a change to a producer or consumer will often require only an adjustment to the messaging layer.

The distributed option would appear to provide the highest level of modifiability, since it not only uses a messaging architecture but also binds control at runtime. It also has the potential to support a heterogeneous environment, depending on the capabilities of the middleware to distinguish among components with varying interfaces/capabilities and allow a RAP or DEVS component to locate an appropriate complementary component on-the-fly. For example, a DEVS model for operating a car could locate a RAP that corresponds to the city in which the car is navigating. This option could even support one-to-many or many-to-many associations, although it is somewhat difficult to find realistic examples.

Although the bridge option does not provide quite the level of dynamic modifiability as the distributed option, the bridge implementation could also be quite extensible at compile-time, if the interface between the bridge and each of the primary components is well-defined and clear boundaries are maintained between the messaging and functional layers.

6 Conclusions

This paper provides an example of a novel architectural problem and shows how it can be analyzed by using architectural attributes. The architect can use the architectural attributes to help define the qualitative measurement of important quality attributes and ultimately select a solution.

The simulation engine/knowledge base system we have looked at has several viable architectural options. One of these, the Knowledge Interchange Broker, has been implemented and illustrates some of the strengths and limitations of this architectural style. The KIB solution supports some timing and ordering assumptions that simplify the complex problem of managing timing issues between two autonomous components.

Future work might include testing an implementation of a RAP/DEVS system using a distributed architecture. In such an architecture, it would be difficult or impossible to control the execution of one component from within the other component, as is the case in the bridge implementation. Therefore, any distributed solution would need to look at other research on timing in order to find ways to support this more flexible architectural option while maintaining the powerful capabilities of the RAP and DEVS components.

7 References

[1] Barbacci, M., M. H. Klein, T. A. Longstaff, C. B. Weinstock, "Quality Attributes." CMU/SEI-95-TR-021, ESC-TR-95-021, December 1995. http:///www.sei.cmu.edu/pub/documents/95.reports/pdf/tr021.95.pdf

[2] Bass, L., M. Klein, and F. Bachmann, "Quality Attribute Design Primitives and the Attribute Driven Design Method", http://www.sei.cmu.edu/plp/bilbao_paper.pdf.

[3] Clements, P., L. Bass, R. Kazman, and G. Abowd, "Predicting Software Quality by Architecture-Level Evaluation", Proceedings, Fifth International Conference on Software Quality, Austin, TX, October 1995. http://ww.sei.cmu.edu/architecture/SAAM.html.

[4] Common Object Request Broker Architecture (CORBA), http://www.corba.org/, 2005.

[5] DEVSJAVA, http://www.acims.arizona.edu, 2005.

[6] Firby, R. J., "Adaptive Execution in Complex Dynamic Worlds", YALEU/CSD/RR #672, January 1989. http://people.cs.uchicago.edu/~firby/papers/thesis/thesis.ps.

[7] Firby, R.J. and W. Fitzgerald, The RAP System Language Manual, Version 2.0, 2000, Neodesic Corporation, Evanston, IL.

[8] Java 2 Enterprise Edition (J2EE), http://java.sun.com/j2ee/, 2005.

[9] Java Native Interface (JNI), http://java.sun.com/j2se/1.4.2/docs/guide/jni/, 2005.

[10] Kazman, R., and L. Bass, "Toward Deriving Software Architectures from Quality Attributes", CMU/SEI-94-TR-10, ESC-TR-94-010, August 1994. http:///www.sei.cmu.edu/pub/documents/94.reports/pdf/tr10.94.pdf.

[11] Sarjoughian, H. S. and J. Plummer, "Design and Implementation of a Bridge between RAP and DEVS", Computer Science and Engineering, Arizona State University, Tempe, AZ, Internal Report, August, 2002, pp. 1-38.

[12] Sarjoughian, H. S., "A Multi-Formalism Modeling Composability Framework: Agent and Discrete-Event Models", The 9th IEEE International Symposium on Distributed Simulation and Real Time Applications, pp. 249-256, Oct., 2005, Montreal, Canada.

[13] Shaw, Mary and Paul Clements, "A Field Guide to Boxology: Preliminary Classification of Architectural Styles for Software Systems", Computer Science Department and Software Engineering Institute, April 1996. http://www-2.cs.cmu.edu/afs/cs.cmu.edu/project/vit/ftp/pdf/Boxology.pdf.

[14] Zeigler, B. P. and H. S. Sarjoughian, "Introduction to DEVS Modeling & Simulation with JAVA: Developing Component-based Simulation Models", January 2005. http://www.acims.arizona.edu/SOFTWARE/devsjava_licensed/CBMSManuscript.zip.

[15] Zeigler, B.P., H. Praehofer, and T.G. Kim, Theory of Modeling and Simulation: Integrating Discrete Event and Continuous Complex Dynamic Systems, Second Edition ed, 2000, Academic Press.