Formalizing the NIST 4-D/RCS Reference Model Architecture Using an Architectural Description Language

C. Dabrowski, H. Huang, E. Messina, J. Horst

December 14, 1999

Contents

Abstract

The 4-D/Real-Time Control System (RCS) Reference Model Architecture provides a well-defined strategy for development of software components for applications in robotics, automated manufacturing, and autonomous vehicles. This architecture has been in the process of definition and evolution for over two decades. To further this work, an investigation has been conducted into the use of Architectural Description Languages (ADLs) as a means to provide a more formal, rigorous definition of the 4-D/RCS Reference Model Architecture and to specify software components for 4-D/RCS systems. ADLs are formally defined languages for specification of software system's designs. Formal specification of system designs is an important precursor to automation of the process of developing software components. In this report, we describe the results of an investigation into the use of ADLs to specify 4-D/RCS software systems, and assess the potential value of ADLs as specification and development tools for RCS domain experts. We conclude that ADLs not only can be used successfully to specify the 4-D/RCS Reference Model, but that they also serve as effective tools to enhance and extend this model. We also find that ADLs potentially can provide a formal basis for automatically checking the consistency of architecture specifications and verifying designs of RCS-based systems against the standardized 4-D/RCS Reference Model. The report discusses prospects for automated reuse of components specified with an ADL, and makes recommendations on improving ADLs as effective tools for specifying, communicating, and validating 4-D/RCS system designs and software components. Finally, the report discusses potential influence of ADLs for commercial software development tools and provides future directions for research.

Chapter 1

Introduction

Architectural Description Languages (ADLs) are specification languages for rigorously describing and analyzing software system designs. This report provides the results of an investigation into the use of ADLs for formally defining the National Institute of Standards and Technology (NIST) 4-D/RCS Reference Model Architecture (Albus, 1997). The project was conducted at NIST under the auspices of the Advanced Technology Program (ATP).

1.1 Motivation and Potential Benefits of ADLs for RCS

The NIST 4-D/RCS Reference Model Architecture provides well-defined guidelines for construction of control software for autonomous real-time systems. The product of over twenty years of research at the NIST Manufacturing Engineering Laboratory, 4-D/RCS prescribes a canonical hierarchical structure comprising intelligent control nodes. This architecture has been widely used for system design of applications in robotics, automated manufacturing, and autonomous vehicles. For example, it has been selected as the software architecture for Department of Defense Demo III eXperimental Unmanned Vehicle (XUV) Program, managed by the Army Research Laboratories (Shoemaker and Bornstein, 1998; Huang et al., 1999).

In recent years, research has progressed steadily to produce a common understanding of the structure and function of 4-D/RCS systems. Efforts have been made to use software engineering disciplines to describe the results in a rigorous manner. These efforts include Object-Oriented approaches (Huang and Messina, 1996), the RCS Generic Shell approach (Huang, 1999), the RCS libraries (Proc-

† **Acknowledgements** The authors wish to express their thanks to John Kenney, David Luckham, and other members of the Stanford University *Rapide* Project for their generous assistance with the *Rapide* ADL and software support tools. Thanks is also provided to NIST staff members who reviewed this paper and the 4-D/RCS prototype specification and provided critical commentary.

tor and Shackleford, 1999), component based software specifications (Messina et al., 1999; Horst et al., 1997), and recently, the Unified Modeling Language (UML).

These efforts provide motivation for developing a formal description of the 4-D/RCS Reference Model Architecture. The use of this Reference Model Architecture as a guideline for system development makes it desirable that any formal description serve as a basis for verifying conformance of application system designs. ADLs are the products of research into computer-processable languages that provide formal description of a software system architecture. A number of existing ADLs support verification of application system designs. Other aspects of ADLs also potentially make them extremely useful in furthering the formalization and continuing evolution of 4-D/RCS systems. In addition, ADLs could provide important capabilities to existing software support tools that automate real-time control system development.

It is important to note that 4-D/RCS is considered a control architecture, and is not purely a software architecture. In this study, we focus on the software aspects of the architecture. To be fully precise, we would use the term *Software* Architectural Description Languages. However, throughout this report, we will use the commonly accepted term Architectural Description Languages to mean those that focus on the software aspects of the architecture.

Commercial tools are available to help implement real-time control systems. They typically provide graphical capabilities for users to design the control system modules, to define the inputs and outputs among them, and to assemble the modules into complete systems. Many of these tools also provide simulation support and have the ability to generate source code for the desired target systems. However, they lack the guidance on how to design a real-time control system. This is analogous to the construction practice. Brick, mortar, wood, and hammers are necessary for constructing a house, but underlying principles of how to design the house and apply the component materials are essential. ADLs have the potential to be married to these software tools to help guide system designers.

1.2 Purpose and Scope of this Report

This report provides the results of an investigation into the applicability of ADLs for specifying and developing the 4-D/RCS Reference Model Architecture. The report:

1. Evaluates the use of ADLs for rigorously specifying the 4-D/RCS Reference Model Architecture and assesses their potential as a means to further develop the Reference Model Architecture and define 4-D/RCS software components.

2. Assesses the use of ADLs as a basis for software support tools that analyze the Reference Model Architecture and system designs, providing

capabilities for simulation, verifying internal system consistency, and verifying conformance of application system designs to the 4-D/RCS Reference Model Architecture.

3. Identifies requirements and future research directions for ADLs from the standpoint of RCS software development, including support for automated component-based software reuse.

4. Assesses the potential benefits of ADLs as useful tools for 4-D/RCS domain experts.

1.3 Method of Study

The goal of the study was to address the issues posed in the previous section. A portion of the 4-D/RCS Reference Model Architecture was selected that most accurately reflected the structural characteristics and functionality of 4-D/RCS systems. This was the Intelligent Control Node, described in Section 2.1.2. All significant ADLs were reviewed in a literature search that identified key language features relevant to RCS. Chapter 3 presents an overview of these salient characteristics of ADLs and identifies assessment criteria used to determine suitability for RCS. Individual ADLs were examined in detail to assess their suitability as specification languages for capturing the structure and function of the 4-D/RCS Control Node. A detailed, comparative analysis of ADL features was not the scope of this study; readers interested in such an analysis should consult (Medvidovic and Taylor, 1999).

A single ADL—*Rapide* (Luckham, 1996)—was selected to construct a prototype specification of a significant portion of the 4-D/RCS Intelligent Control Node. Resources were not available to develop additional prototypes in other ADLs. The *Rapide* language is described in Section 3.2. Chapter 4 describes the prototype specification. The draft *Rapide* specification is included in Appendix A. The specific aspects of *Rapide* used in developing the prototype specification were compared with features offered by the other ADLs. The conclusions, presented in Chapter 5, provide a basis for both identifying requirements for ADLs to specify 4-D/RCS software architectures and components as well as recommending future research directions for ADLs.

Chapter 2

The RCS Reference Architecture

2.1 Overview of RCS Concepts

Developed over the course of two decades at the National Institute of Standards and Technology and elsewhere, RCS has been applied to multiple and diverse systems (Albus, 1995). In addition to the Army Demo III program, RCS application examples include coal mining automation (Horst, 1993), the NBS/NASA Standard Reference Model Architecture for the Space Station Telerobotic Servicer (NASREM) (Albus et al., 1989), a control system for Multiple Autonomous Undersea Vehicles (Albus and Blidberg, 1987), and a control system for a U.S. Postal Service Automated Stamp Distribution Center (USPS, 1991). There are numerous manufacturing applications, including the Open Architecture Enhanced Machine Controller (Albus and Lumia, 1994) and testbeds within the National Advanced Manufacturing Testbed facility at NIST including an Inspection Workstation and a welding cell.

2.1.1 RCS as a Reference Model Architecture

RCS provides a reference architecture and an engineering methodology to aid designers of complex control systems. Guidelines are provided for decomposition of the problem into a set of control nodes, which are arranged hierarchically. The decomposition into levels of the hierarchy is guided by control theory, taking into account system response times and other factors, such as planning horizons. RCS is focussed primarily on the real-time control domain. It can be further specialized into application-specific versions. 4-D/RCS (Albus, 1997) is one such version, which is aimed at the design and implementation of control systems for intelligent autonomous vehicles for military scout missions. The "4-D" portion of the name comes from the integration of the work by the German Universitat der Bundeswehr - Munchen VaMoRs approach to dynamic machine vision, in which three spatial dimensions and time are tracked (Dickmanns et al., 1994). This particular flavor of RCS was studied with respect to ADLs.

2.1.2 The RCS Intelligent Control Node

RCS is based on the generalization of principles of intelligent processing and control that are exhibited by human reasoning processes and behaviors, manufacturing systems, and military command structures. These principles form the basis for a "building block" approach to designing and implementing systems within RCS. A basic set of functions is defined to exist for each RCS control node, which follow a "sense-model-act" paradigm. These functions were found to be necessary (Albus, 1991) in order to accomplish complex behaviors. The model therefore enables intelligent control. We define intelligent control as that which causes a complex system to successfully perform complex physical tasks in the presence of uncertainty and unpredictability. The RCS approach has the further benefit of providing a framework in which to design and build the software for intelligent control. Adopting a standard set of functions, communications pathways, and interface specifications, along with the software decomposition guidelines, provides a strong basis for facilitating development and promoting reuse. Figure 2.1 depicts the model for an RCS control node.

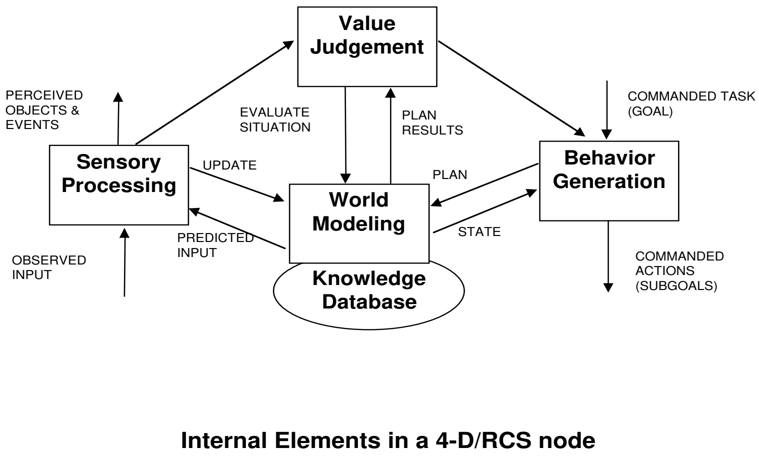

Figure 2.1: Model for an RCS Control Node.

The model for an RCS control node contains a behavior generation (BG) function that makes decisions based on the received task commands and on the current state of the world. BG is supported by world modeling (WM), value judgement (VJ), and sensory processing (SP) functions (Albus and Meystel, 1996; Barbera et al., 1984). System developers insert the appropriate algorithms into the BG, WM, VJ, and SP modules. BG consists of job assignment, planning, and control functionality. These functions are the focus of the ADL prototype developed as part of this study. The WM modules contain information about the state of the problem domain. WM may provide simulation facilities to estimate the state of the world at present or some future time. WM can be used to answer questions about the outcomes of performing certain plans. WM also contains longer-term information in a Knowledge Database (KD). KD includes symbols and data structures containing information about entities, events, and how the world behaves. VJ computes costs, risks, and benefits, and evaluates the relative merits of certain courses of action, as may be simulated via WM. SP modules process data from proprioreceptive, visual, auditory, and other sensors. The sensed information is filtered, differenced, and correlated in order to extract information about the environment and the system itself. Feature extraction, pattern recognition, and data fusion are typical SP functions.

2.1.3 The Behavior Generation Module

Behavior Generation (BG), shown in further detail in Figure 2.2, is at present the most fully developed element of RCS, hence the functionality of this component provided ample material to develop the prototype specification described in Chapter 4.

2.1.4 The 4-D/RCS Seven Level Hierarchy

The BG module parallels the planning and execution within a hierarchical organization, with a superior assigning tasks to its subordinates in order to accomplish the desired goals. Distinct sub-modules exist within the BG module. They consist of a Job_Assignor (JA), a set of plan action Schedulers (SC), a Plan Selector (PS), and a set of control Executors (EX). The Job_Assignor (JA) decomposes input tasks into job assignments for each subordinate to the node. The Scheduler (SC) accepts a job assignment and computes a schedule for its subordinate. There is an SC submodule for each subordinate to the node. JA and SC produce a tentative plan that uses the resources available to the node (including its immediate subordinates) to accomplish the commanded task. The Plan Selector (PS) works with the WM plan simulator and the VJ plan evaluator to select the best overall plan for each of the subordinates. There is an Executor (EX) corresponding to each subordinate. Each EX executes its portion of the selected plan, coordinating actions between subordinates, and correcting for errors between the plan and the evolution of the world state reported by the world model, effectively closing the loop between command and feedback.

The 4-D/RCS reference model architecture for intelligent, real-time control of autonomous systems is composed of a number of intelligent control nodes that work together to perform complex tasks. Figure 2.3 is a high-level diagram depicting a portion of the 4-D/RCS hierarchy for an autonomous vehicle.

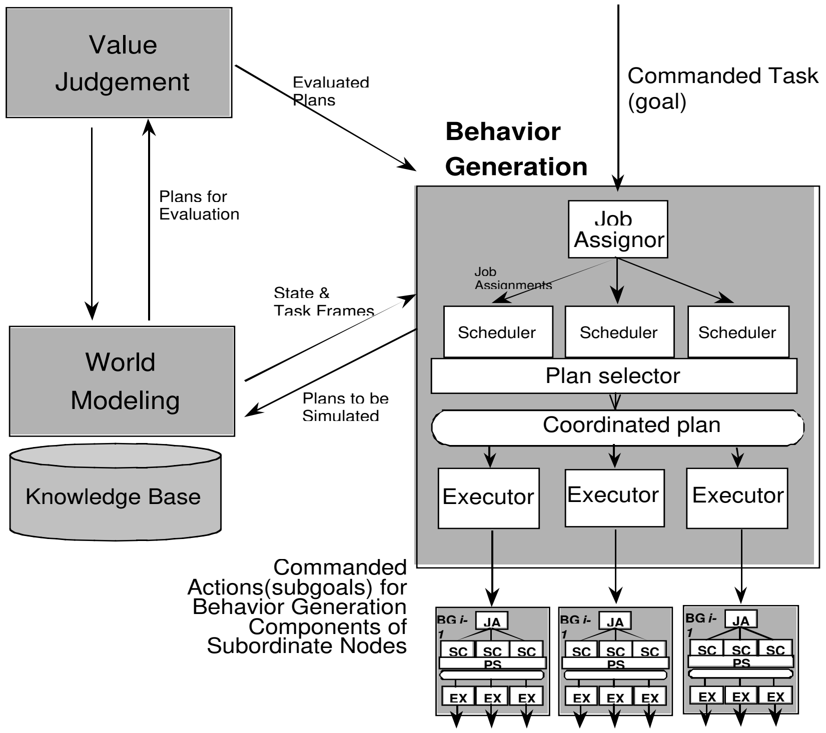

Figure 2.2: Details of the Behavior Generation Module.

Each node in the hierarchy is built upon the SP-WM-BG-VJ internal elements shown in Figure 2.1. There are seven levels in the vehicle's control hierarchy. Control nodes at one level decompose tasks into smaller units of work, which are assigned to individual control nodes at the next lower level. The upper levels (battalion, platoon, and section) correspond to the military command chain and are resident on the vehicle itself to be invoked when higher level decisions are required and the vehicle is out of contact with its superiors. Hence the term "surrogate."

Each level is designed to function in a particular spatial and temporal scope. The temporal scope is based on the response times required for control. The spatial scope for each level corresponds to the planning horizon required. The time scale increases, typically by tenfold, at each higher level in the hierarchy, as do spatial extents. The response times for the levels are correspondingly slower. The spatial resolution is also correspondingly coarser at higher levels. Thus, the complexity level is constant throughout the hierarchy. Figure 2.3 shows example temporal and spatial scopes for each level. The servo level converts component commands to actuator coordinates and does not typically use a spatial map.

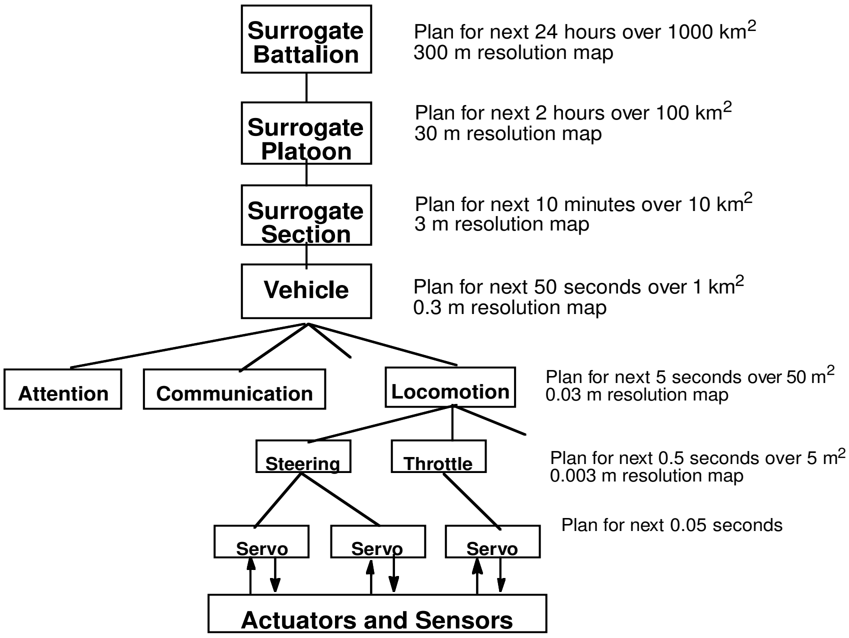

Figure 2.3: Example 4-D/RCS Hierarchy for an Autonomous Scout Vehicle

2.1.5 Current State of Software Engineering for RCS

Construction of RCS-based systems is based on the canonical hierarchical structure of RCS. All the RCS control nodes contain basic elements of decomposition (BG, SP, WM, and VJ) and common processing or bookkeeping functions. The control nodes follow a consistent pattern to process their input commands. RCS defines a command and status execution model with standard interface channels and messages, for which software templates (base classes) have been developed (Huang et al., 1999). Developers can use these templates as a starting point for building their applications. The developers specify the actual interface channels and add application-specific messages. They add their particular algorithms within the provided slots in the corresponding module. A set of software tools and libraries is under development at NIST to support the software development of RCS systems (Shackleford and Proctor, 1998).

Although some tools and libraries are available, better communication of RCS concepts and greater reuse continue to be desirable. The primary means of communication of RCS reference architecture and methodology remains English language description. Methodology and tool-independent diagrams are also used to convey concepts, such as the job assignment, scheduling, and plan selection functions of Behavior Generation shown in Figure 2.2. Work is underway in the formalization of RCS component specifications in order to facilitate searching for reusable algorithms or pieces of software (Horst et al., 1997). The specification research is part of a long-term effort aimed at achieving automated software composition of systems from components. Given a formal specification of a system's requirements, an automated composition tool could find components that match the requirements by checking the component's specifications and assemble the desired system automatically. The study presented in this paper follows along the lines of that work by providing an ADL specification for the Behavior Generation Module, discussed in more detail in Chapter 4.

Some aspects of RCS lend themselves naturally to adopting an object-oriented representation. Since the design of a system under the RCS architecture focusses on physical devices and equipment, it has a natural affinity to object-orientation. A primary driver of the system decomposition during an RCS design phase flows from the concrete "objects" in the system, such as motors, wheels, and brakes, that receive and act upon commands or messages. A conceptual comparison of RCS and object-oriented methodologies was published (Huang and Messina, 1996) in which several similarities were observed. However, the RCS approach for system analysis and design was found to have certain aspects that distinguished it from classical object-oriented techniques. Primarily, RCS has a much greater focus on behavior analysis. RCS also requires temporal and spatial decomposition as part of system analysis and design. Beyond the contrasts noted in the Huang and Messina paper, there are other areas where object-oriented approaches may not provide software engineering support desirable for RCS-based development. These include full real-time support, analysis and verification for the architecture or implementation of the architecture, execution or simulation capabilities based on the design specification, and conformance validation.

NIST researchers have recently started investigating the use of UML to describe RCS and to model RCS applications. This effort includes applying the sequence diagram and the collaboration diagram concepts to model the system behavior. While we have obtained early positive results, some of the other observations that we made on the object-oriented representations still apply to UML. This is due to the fact that a significant portion of the UML language is based on the object-oriented paradigms.

Gaps in the support provided by object-oriented tools and methodologies further stimulated interest in the potential of ADLs. For instance, UML provides no support for real time systems and no direct support for modeling software architectures. Object-oriented methods in general are data-centric, providing only for some generic behavior description capabilities. Analysis of the architecture and simulation of the execution of an architecture are not possible in most object-oriented tools. These gaps are recognized by the UML community and are leading to evolution in the modeling language. For example, The Object Management Group recently issued a Request for Proposals under the title "UML Profile for Scheduling, Performance and Time" (OMG, 1999). This proposal is aimed at expanding the UML to include support for modeling of time-related paradigms, which are essential for the design and specification of real-time systems.

Chapter 3

Architectural Description Languages

3.1 Overview of Architectural Description Languages (ADLs)

Garlan and Perry define a software architecture as consisting of the "structure of the components of a program/system, their interrelationships, and principles and guidelines governing their design and evolution over time" (Garlan and Perry, 1995). A software architecture can be seen as an abstract system specification of the system's functional components, their behavior, their external interfaces, and interconnections with other components. A specification is created using a language that is composed of a set of rigorously defined keywords and operators coupled by a defined grammar (Feijs, and Jonkers, 1992). An ADL is "a language that provides features for modeling a software system's conceptual architecture" (Medvidovic and Taylor 1999).

3.1.1 The Concept of Abstract Specification of Software System Designs

Creating an abstract specification of a software system means describing only the essential aspects of the system and omitting other details. An abstract specification may be limited to identifying the components that compose the system, component inputs and outputs, and any other aspects needed to make the specification usable. Only a partial description of the computation performed by the component would be provided, or this description could be omitted altogether. The complete specification, including code, is worked out later by actual software developers. By describing only the most essential elements, an abstract specification reduces the size and complexity of the description, making it more manageable and allowing easier comprehension. In this way, the abstracted description of the entire design or its components can be reused many times to develop different systems that share the same basic structure. In principle, an abstract specification is *implementation neutral*, i.e., it can be implemented in more than one programming language such as C++, PL1, or Java.

3.1.2 Generic ADL Features for Specifying Software Architectures

ADLs provide language constructs for specifying the software system components, the connections between those components, and the overall structure of the system or its topology. A generic set of ADL capabilities has been identified in (Garlan and Shaw, 1994; Medvidovic and Taylor, 1999; and Vestal, 1993). These capabilities are summarized below.

<u>Specifying Software Components</u>. ADLs commonly describe software components by defining interfaces for them. An interface definition may include a *signature*, or description of the messages and commands accepted and sent by a component together with arguments and outputs (results). The signature may be accompanied by constraints upon the messages, such as the order in which they must be sent or received, or upon the values that arguments may have. In addition, a description of the behavior of the component in response to externally (or internally) generated messages may be provided either as part of the definition of the component's interface or its internal description. Specification of behavior is described further below.

<u>Specifying Connections Between Components and Software Architectures</u>. ADLs support definition of constrained *connections* between messages of different components. Connections specify which components receive the messages emitted by other components, thus defining a sort of pipe between components. An *architecture* is, at a minimum, a description of a set of components or their interfaces, together with the connections between those components. An architecture may define constraints on the connections, in which case argument values are restricted or messages are prescribed to occur in a particular sequence.

<u>Specifying System Behavior Using an Underlying Computational Model</u>. Some, though not all, ADLs support specification of computations performed by a system, referred to as system behavior. Usually, an ADL employs a formally defined descriptive method or underlying *computational* model to provide the necessary semantics. Constraints on behavior are also defined in terms of the computational model. Examples of computational models are Finite State Machines (FSM), Communicating Sequential Processes (CSP) and Partially-Ordered Sets of Events (POSETS), though the formalisms employed vary widely among different ADLs. An ADL may allow description of the behavior of component interfaces, component internals, and component connections.

<u>Defining Architectural Styles</u>. *Architectural styles* (Shaw, 1994; Allen, 1997) provide rules or constraints that place limitations on how components may be connected and on what system topologies may be described. One example of an *architectural style* is a pipeline architecture in which components are connected in a sequence so that the output of one component becomes the

input to at most, one other. Another example is a top-down hierarchical style in which data flows from a single central node to sets of subordinates. The 4-D/RCS Reference Model Architecture is an instance of a top-down hierarchical architecture. Being able to explicitly declare an architectural style allows the specification to be defined more formally and constrained more precisely. It also provides an additional means for judging whether or not a particular application system design conforms to a Reference Architecture—an important consideration for 4-D/RCS domain experts. Some, but not all, ADLs allow explicit declaration of architectural styles.

3.1.3 Differences Between ADLs and Programming Languages

Like a programming language, an ADL provides a well-defined syntax and semantics. Unlike a programming language, an ADL provides a more abstract, partial specification of a system. The major objects that can be explicitly defined in an ADL—components, connections, and architectures—are first-class objects. This means that these objects may be declared as types, instantiated, subtyped, or passed as parameters and manipulated in the same way a programming language manipulates structured data types such as arrays or records. The use of a computational model as an underlying basis allows behavior to be described at a more abstract level using restricted, well-defined semantics. For instance, in the *Rapide* ADL (Luckham, 1996), behavior is described in terms of events, event dependencies, and partial orderings of events (described further below).

3.1.4 Use of ADLs for Analysis and Verification of Software System Designs

The use of a well defined, rigorous specification language provides a basis for formal analysis of a specification and the verification of software system designs. Some ADLs employ formal proof techniques to determine whether desirable properties, such as internal consistency, hold within a specification. Analysis of ADL specifications may also take place through simulation support tools, which allow the specification to be executed and a result to be computed, thus simulating the computations to be performed by the system being specified. System developers may then either review an animated simulation interactively or analyze partial results using automated support tools. An important research area for ADLs is the development of analysis capabilities for rigorous comparison of specifications to verify system designs. Support for this form of verification is of significant interest to the 4-D/RCS community, which has a long-standing interest in verifying conformance of individual system application designs to the canonical 4-D/RCS Reference Model Architecture.

3.1.5 ADL Evaluation Criteria Used in the Study

The study evaluated the applicability to 4-D/RCS of the ADL features described in the preceding sections. It was important that ADLs be able to represent the structural aspects of RCS software systems, including RCS modules, module interactions, and the top-down decomposition structure of RCS systems. A type system that allowed definition of the complex data types needed by RCS was also critical. To model system behavior and internal component semantics, it was necessary to determine if ADLs could represent the computational models needed for the real-time processing required by RCS.

To meet the goal of formalizing the Reference Model Architecture, both the ADL and the specifications produced by an ADL had to be rigorous. The extent to which ADLs provided a formal basis for description of architecture, components, and system behavior determined the degree to which it was possible to define analysis functions that could be applied to a specification by a software tool. The evaluation of the potential of ADLs as languages on which to base software development and analysis support tools was particularly critical, because the RCS requirement for software development tools is substantial. A closely related goal of this study was to determine how ADL research might impact existing commercial tools for development of real-time systems. These evaluations required in-depth examination and the development of a prototype specification using an ADL. This is described in the next two sections. Chapter 5 provides the conclusions of the study.

3.2 The *Rapide* ADL

Rapide (Luckham, 1996) is an ADL and supporting tool set developed at Stanford University in the mid-1990s. This ADL was chosen as the primary focus of this study because of its well-developed capability for representing and simulating real-time system designs.

3.2.1 Rapide Language Features

Rapide supports most of the features described above that are common to ADLs. *Rapide* permits definition of a set of component *interface* types each of which has a signature that includes events generated and received by components of that type and (optionally) a description of the component's behavior. An interface may also define *constraints* that require dependencies between events, place limitations on the order of events, constrain parameter values, or make other limitations. The internal details of the components themselves—known as *modules*—may also be specified. A module description specifies internal behavior and supporting data structures that allow the module to conform to its *interface*. That is, the internal behavior of the module is defined so that it responds to events received by the interface, generates events sent by the interface, and conforms to any constraints defined in the interface.

In a specification of a *Rapide* software architecture created by a user, *interfaces* and *modules* are used to specify system components. The software architecture is formally described by connecting types of events generated in one *interface* specification to events received by another *interface*. A *module* conforming to an *interface* may be decomposed into a sub-architecture consisting of a set of connected component *interfaces*†.

Connections between types of events of different *interfaces* and the specification of a component's behavior define causal dependencies of the events. During the simulated execution of a software architecture, these dependencies can be aggregated to form POSETs, or partially ordered set of events. These aspects are described in detail in the remainder of this section. Examples are provided in Chapter 4.

3.2.2 The *Rapide* Computational Model

The focus of *Rapide* is the definition of software architectures for real-time systems. *Rapide* provides a sophisticated language for defining event types, event causality, and behavioral constraints. Within the definitions of *interfaces* and in the definition of component *connections*, event types are defined to trigger, or cause, other events. Thus, the behavior of an interface (or its underlying module) may be defined to send an event to another interface through a connection which, when received, causes the generation of additional events. This is depicted graphically below in Figure 3.1.

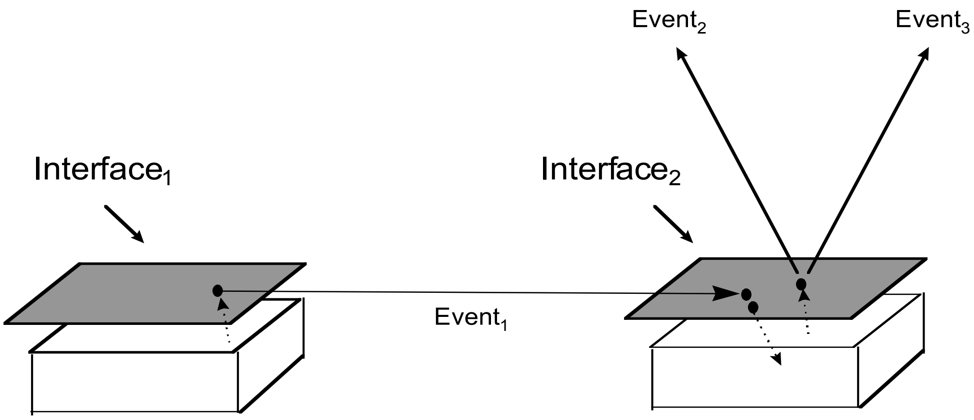

Figure 3.1: Graphical Depiction of Interfaces and Events.

Rapide also provides the ability to express time and timing constraints. Insufficient time and resources were available to explore this capability during this

†Unlike some ADLs, connections are not explicitly defined as first-class objects in *Rapide*, but are specified directly as part of the architecture. *Rapide* also does not provide explicitly for the definition of architectural styles.

study.

Example of a *Rapide* Interface. In Figure 3.1, Interfaces are depicted as shaded areas or two-dimensional planes while underlying modules are shown below as boxes. Dashed arrows between interfaces and modules indicate modules receiving or generating events in conformance to the architecture. Events between interfaces are shown by solid lines.

A *Rapide* specification of "Interface$_2$" in this example might be written as follows:

TYPE Interface$_2$ IS INTERFACE;
ACTION
IN
    Event$_1$;
OUT
    Event$_2$;
    Event$_3$;

BEHAVIOR
    Event$_1$ ||> Event$_i$;
    Event$_k$ ||> Event$_2$;

CONSTRAINT
    NEVER Event$_1$ || Event$_2$;
END;

POSETs and Simulated Execution of an Architecture. The specification shows "Interface$_2$" defined as an interface type containing implicit event type definitions. In the "Interface$_2$" definition, "Event$_1$" is a received event type while "Event$_2$" and "Event$_3$" are generated event types. The behavior description in the definition shows "Event$_1$" as causing "Event$_i$" to occur, denoted by the "||>" symbol. "Event$_i$" may be handled by the underlying conforming module that performs a computation and generates another event, "Event$_k$". The occurrence of "Event$_k$" causes "Event$_2$" to be transmitted through the interface. A *constraint* on this interface is defined which states that "Event$_1$" and "Event$_2$" must never be independent, where independence is denoted by "||". This means "Event$_1$" and "Event$_2$" must always be causally connected in a POSET. A similar set of causal relationships could be defined between "Event$_1$" and "Event$_3$".

An event is said to be causally dependent on all events that either directly result in its generation or in the generation of its predecessors. It is considered independent of all other events. In actual *Rapide* specifications of architectures, very large causal sequences of event types and event constraints can be defined both in interface definitions and as part of connections between interfaces. The causal sequences serve as a basis for "executing" a specification using *Rapide* software support tools to produce simulations. During the simulation, the event types defined in the specification result in instances that execute according to their defined behavior. The execution of a software architecture specification produces a partially ordered set of event instances, called a POSET, which describes the generated events together with their causal dependencies.

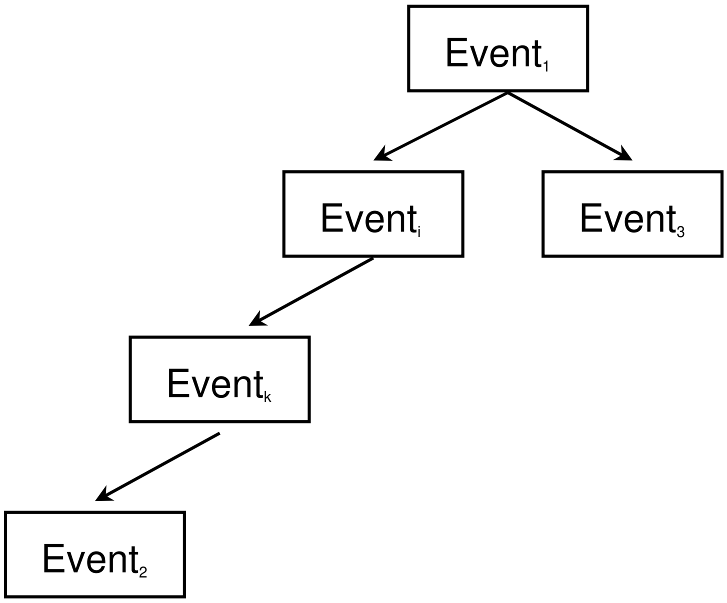

Figure 3.2: Depiction of a Simple *Rapide* POSET

The sample POSET in Figure 3.2 shows a causal sequence consisting of Event_1, Event_i, Event_k and Event_2 which depicts the situation shown in the interface definition above. However, note that Event_3 is not dependent on (e.g., is independent of) Event_2.

In *Rapide*, significantly more complex system behavior may be defined in which special operators can be used to create independent, parallel sequences of events that can be either deterministic or non-deterministic (Luckham, 1996). The resulting POSETs can be quite elaborate and almost unlimited in size.

3.2.3 Conformance

Rapide provides a capability for verifying that the behavior of an application system design, called *concrete* architecture, conforms to that of a more *abstract* architecture, such as the 4-D/RCS Reference Model Architecture. This is accomplished by first declaring a set of constraints in the abstract architecture and then declaring an equivalence, or mapping, of events from the *concrete* to the *abstract* architecture. The mapping of events from the *concrete* to the *abstract* architecture may be many-to-one. The *abstract* architecture is then executed with the "mapped" events of the *concrete* architecture mapped onto it. Conformance to constraints of the *abstract* architecture is tested. If constraints are violated, an error message appears and the concrete architecture can be deemed as non-conformant.

3.2.4 The *Rapide* Toolset

In *Rapide* the POSET is the basis for automated analysis conducted by an associated toolset. A *Rapide* specification may be defined using the RAPARCH tool, which has a graphical front-end, to specify interfaces and interface connections in a software architecture. A complete specification in the *Rapide* language is translated into a C++ program, which when compiled and executed, produces a POSET for the defined architecture.

Rapide provides a simulation tool called RAPTOR for producing an interactive graphical animation of the execution of the specification in which interfaces and connections are depicted as icons while event icons move between interfaces. The POSET Viewer, or POV, gives a static picture of a POSET with events and causal arrows between events. Query functions can be used to select interesting subsets of the POSET and provide detailed information. A method is provided for verifying system designs against a more general Reference Model architecture based on comparison of POSETs.

Chapter 4

The Experiment

4.1 Specifying a 4-D/ RCS Control Node in *Rapide*

In order to help answer the questions about the applicability of ADLs to 4-D/RCS set forth in Chapter 1, the *Rapide* ADL was used to specify a large piece of the 4-D/RCS Intelligent Control Node. The basis for the Control Node specification was NISTIR 5994, *4-D/RCS: A Reference Model Architecture for Demo III* (Albus, 1997). The specification was developed by two of the coauthors: one focusing on the study of ADLs; and the other, a domain expert in design of 4-D/RCS systems who regularly reviewed the model and guided its evolution. The specification was reviewed and verified by a larger group of experts in 4-D/RCS. In addition, the use of an ADL to ascertain conformance of individual system designs to the Reference Model Architecture was examined. Conclusions reached about the use of ADLs for RCS are provided in Chapter 5.

4.1.1 Overview of the Prototype Specification

Component interfaces were defined for each 4-D/RCS Intelligent Control Node module together with the events handled, sent, and received and applicable constraints for the module. The specification provided the decomposition of the Control Node into its major subcomponents: Behavior Generation (BG), World Modeling (WM), Value Judgement (VJ), and Sensory Processing (SP). Behavior Generation was further decomposed into Job_Assignor (JA), a set of Schedulers (SC), a set of Executors (EX), and a Plan Selector (PS). World Modeling (WM) was decomposed into its Simulation and Knowledge Base components. The architecture specification included the connections between the interfaces defined for the modules. A sufficient amount of behavior was included to allow the architecture to be simulated using the *Rapide* toolset. The entire specification encompassed more that 1000 lines of *Rapide* code, which is included in Appendix A.

4.1.2 Details of Job_Assignor and Scheduler Functions

The use of ADLs to specify 4-D/RCS is illustrated in a sample *Rapide* description of the interaction of two subcomponents of the 4-D/RCS Behavior Generation Module: The Job_Assignor and a Scheduler (of which there may be several instances). The conceptual design for this representative fragment of the Reference Model functionality, described in Chapter 2, is shown graphically in Figure 4.1. The fragment contains only a subset of the actual events and behavior defined for these components. The specifications of algorithms for computing schedules and selecting plans in underlying modules are omitted from the Reference Model Architecture because they are application-specific.

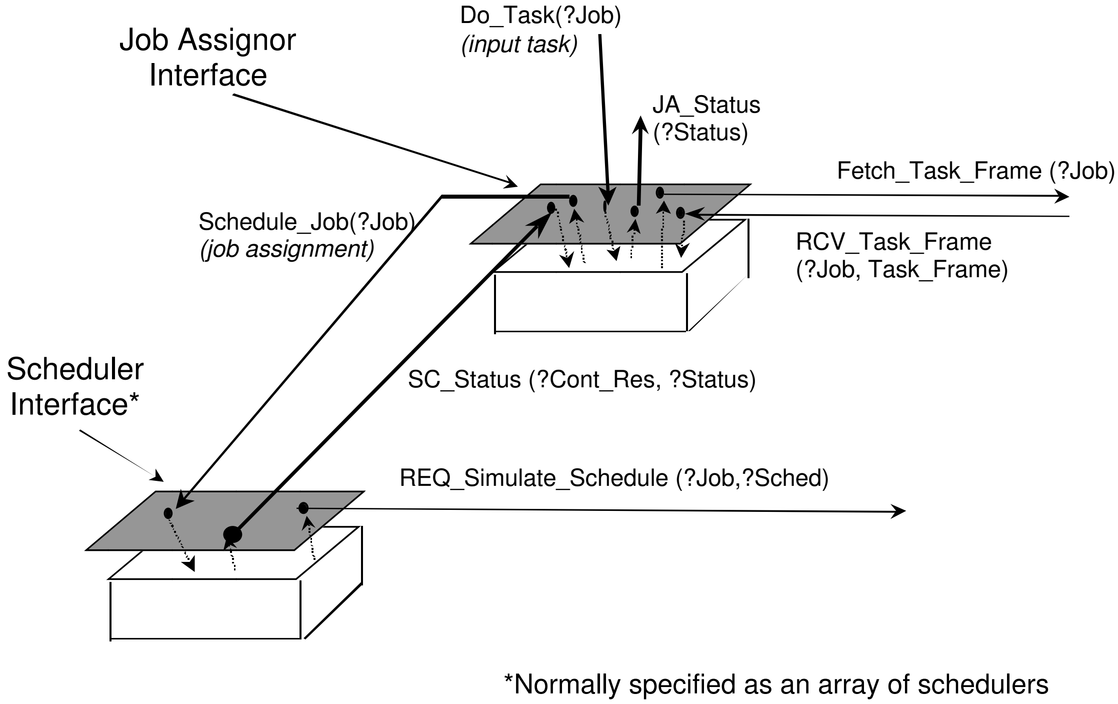

Figure 4.1: Job_Assignor and Schedulers in the Behavior Generation Module

The graphical notation from Figure 3.1 is supplemented by the use of variable arguments for events denoted by ?Task, ?Job, or ?Status. Figure 4.1 shows a *Job_Assignor* component defined as a *Rapide* interface. The *Job_Assignor* interface signature receives a Do_Task event representing an input task in which ?Task is the argument variable for a task name. The *Job_ Assignor* generates a *Fetch_task_frame* event with the job name as an argument that is passed outside *Behavior_Generation* to the *World_Modeling* module. *World_Modeling* returns a task frame data structure containing information necessary to perform the task that is received by *Job_Assignor* as a *RCV_Task_Frame* event. The underlying module for *Job_ Assignor* decomposes the task frame into job assignments (process not shown) for the schedulers. Figure 4.1 depicts the generation of a *Schedule_Job* event, representing a job assignment to the *Scheduler* interface.

The *Scheduler* receives the *Schedule_Job* event. Its underlying module computes a schedule, which is transmitted as an event through the interface outside of *Behavior_Generation* to *World_Modeling* plan simulator. This is depicted as a plan in Figure 2.1, "Model for an RCS Control Node." Ultimately, the simulated plans are evaluated by *Value Judgement* and returned to the *Plan Selector* in the *Behavior Generation* module (described in Chapter 2 but not shown in the example). The *Scheduler* interface is also shown as returning a Status event with a ?Status variable. Values for specific status events would be generated in underlying modules that conform to the interface.

4.1.3 Specification of the Interfaces, Behavior, and Constraints

A partial *Rapide* specification of the *Job_Assignor* interface is given below. The *Job_Assignor* is declared to be of type Interface. The signatures for the events received by, and sent from, this interface are provided including variable arguments and their types.

```
TYPE Job_Assignor_Interface IS INTERFACE;
ACTION
    IN
        Do_Task (Task : Task_Command_Frame),
        RCV_task_frame (Task : Task_Command_Frame; TF : Task_Frame),
        SC_Status (CR : Controlled_Resources; ST : String);
    OUT
        Schedule_Job (Job : Task_Command_Frame),
        Fetch_task_frame (Task : Task_Command_Frame),
        Decompose_task_frame (TF : Task_Frame),
        JA_Status (?status);
BEHAVIOR
    (?Task : Task_Command_Frame)
     Do_Task (?Task) ||> Fetch_task_frame (?Task);
    (?Task : Task_Command_Frame; ?TF : Task_Frame)
     RCV_task_frame (?Task, ?TF) ||> Decompose_task_frame(?TF);;
END;
```

A portion of the behavior depicted in Figure 4.1 is also specified. The receipt of a Do_Task command to perform a task triggers a request for a task frame containing essential information needed to perform the task. A causal connection is defined between these two events. The (?Task) is a variable placeholder that denotes the task. Similarly the receipt of a RCV_task_frame command results in a Decompose_task_frame in which (?TF) denotes the variable placeholder for the task frame which is transferred. The Schedule_Job and Status events are generated through the interface by underlying conforming modules which also instantiate the necessary arguments. These are omitted from this portion of the specification example.

The specification of the Job_Assignor is supplemented by the declaration of constraints shown below.

```
CONSTRAINT
–
– (C1) Do not allow causally independent
– Do_Task and Schedule_Job events!
–
NEVER (?Task : String; ?Job : String)
        Do_Task (?Task) || Schedule_Job (?Job);
–
– (C2) Do not allow causally independent
– Do_Task and Status Message events!
–
NEVER (?Task : String; ?status : String)
        Do_Task (?Task) || JA_Status (?status);
```

Constraint C1 prohibits the independence of Do_Task and Schedule_Job events, while constraint C2 prohibits independence of Do_Task and Status_Events. These constraints require that that these events *must always be* related in a causal sequence.

A partial specification of the Scheduler interface is given below:

```
TYPE Scheduler_Interface IS INTERFACE;
ACTION
    IN
        RCV_Schedule_Job (JOB : Task_Command_Frame),
        ................;
    OUT
        SC_Status (Cont_Res : Controlled_Resources; ST : String),
        REQ_Simulate_Schedule (CR : Controlled_Resources;
            Job : Task_Command_Frame Sched : Schedule),
            ................;
END;
```

The signature declaration shows the Schedule_Job as a received event and the REQ_Simulate_Schedule and SC_Status as transmitted events. The behavior for computing schedules and determining status would be implemented in application-specific modules.

4.1.4 Specification of the Architecture

The specification of the portion of the Behavior Generation architecture from Figure 4.1 is given below. This specification shows the connection of the events between the Job_Assignor, an array of Schedulers and the Plan Selector.

```
ARCHITECTURE BG_Module_Arch () .........
IS
        JA : Job_Assignor_Interface IS Job_Assignor_Module();
        SC : array [integer] of Scheduler_Interface IS (1.. $Num_Controlled_Resources,
        ..........)
        PS : Plan_Selector_Interface IS Plan_Selector_Module();
        ..........
CONNECT

        (?Job : Task_Command_Frame)
        JA.Schedule_Job(?Job) ||> SC i.RCV_Schedule_Job(?Job);

        (?CR : Controlled_Resources; ?ST : String)
        SC[ i].SC_Status (?CR, ?ST) ||>
        JA.SC_Status (?CR, ?ST);
        ........................
        (?CR : Controlled_Resources; ?Job : Task_Command_Frame;
        ?Sched : Schedule; ?ST :string)
        PS.SND_PS_Status (?CR, ?Job, ?Sched, ?ST) ||>
                SC[i].RCV_PS_Status (?Job, ?Sched, ?ST);
```

Note that each of these components is first declared as an instance of one of the types defined above. This is followed by explicit connections between OUT events in the interface of one component and IN events declared in another interface. The CONNECT keyword is used to establish the relationships between outputs and inputs of interfaces. The *Rapid*e symbol "||>" is used to indicate a causal connection between these events. For example, the Schedule_Job event emitted by the *Job_Assignor* is sent to and received by the *Scheduler*, also as a Schedule_Job event.

4.1.5 Execution of the 4-D/RCS Intelligent Control Node Architecture

The declaration of causal connections between events in *Rapide interfaces* and in the declaration of the architectures defines a causal sequence of events. The execution of this architecture produces the POSET shown in Figure 4.2, which omits intervening events not described in the partial specifications given above.

The figure shows the causal connection between the Do_Task event and a Fetch_Task_Frame event that retrieves information necessary to initiate scheduling activity in a control node. When the Job_Assignor receives the Task Frame,

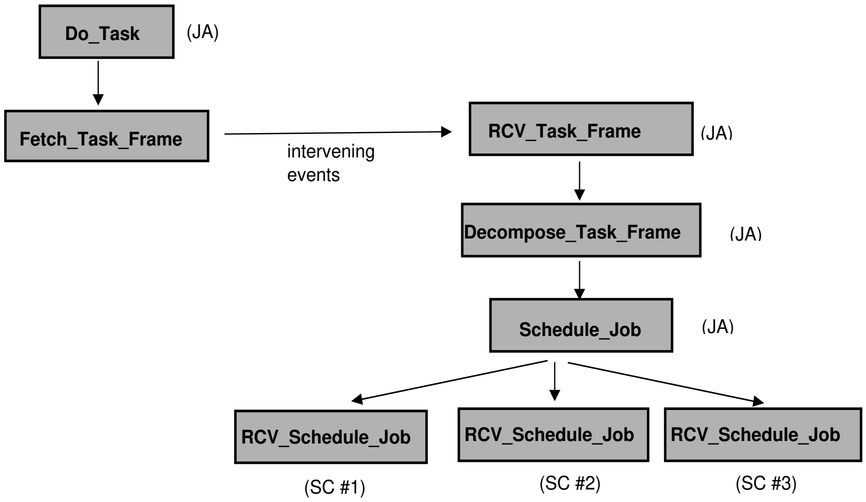

Figure 4.2: Event trace of *Rapide* Reference Model Specification

this triggers the Decompose_Task_Frame event, followed by the Schedule_Job event that is forwarded to a set of Schedulers. This example could be extended to incorporate sophisticated *Rapide* capabilities for representing parallelism, non-determinism, and time together with the resulting POSETs. While of great interest for modeling 4-D/RCS systems in general (see Section 5.2.4), specific processes that involve these properties are not defined at the level of the Reference Model Architecture.

4.1.6 Verification of Individual System Designs Against the Reference Model Architecture

Rapide provides a capability for verifying that the behavior of a system design, or *concrete* architecture, conforms to that of a more *abstract* architecture, such as the 4-D/RCS Reference Model Architecture. This is accomplished by first declaring a set of constraints in the abstract architecture and then declaring an equivalence, or mapping, of events from the *concrete* to the *abstract* architecture. The mapping of events from the *concrete* to the *abstract* architecture is explicitly defined in a specification (see example below) and may be one-to-one or many-to-one. The *abstrac*t architecture is then executed with the "mapped" events of the *concrete* architecture replacing events originally defined in the *abstract* architecture. Conformance to constraints of the *abstrac*t architecture is tested. If constraints are violated, an error message appears and the concrete architecture can be deemed as non-conformant.

An example of a Job_Assignor System Design. An example of this approach is described in which the Job_Assignor definition in Section 4.1.3 is a component of the abstract Reference Model Architecture. It has two previously defined constraints: C1 and C2. A concrete Job_Assignor Interface that is part of an application system design is declared below.

```
TYPE Job_Assignor_Interface_App IS INTERFACE
        ACTION
                IN Do_Task_App (task_command_frame : String),
                OUT Schedule_Job_App (task_command_frame : String),
                        JA_Status_App (?ST : string);
END;
```

The interface definition is accompanied by the specification of an underlying module. This specification, though it conforms to the interface, generates parallel Do_Task events that are independent of Schedule_Job and Status events.

```
MODULE Simulate_Events_JA_Bad()
RETURN Job_Assignor_Interface_Bad IS
PARALLEL
        Do_Task_Bad ("Task #1");
||
        Schedule_Job_Bad ("Job #1a");
        JA_Status_Bad ("Done");
END;
```

The execution of this specification would result in the following POSET:

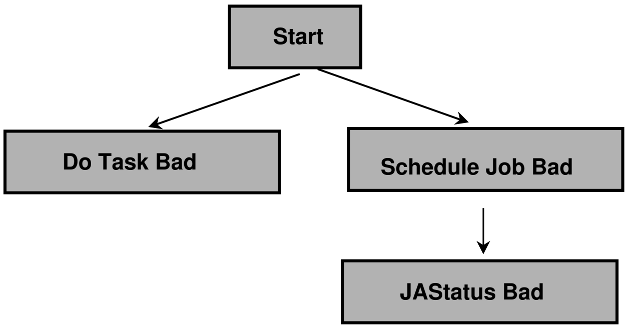

Figure 4.3: POSET for non-conforming Job_Assignor

<u>Mapping the Job_Assignor System Design to the Reference Model</u>. To establish that this non-conforming Job_Assignor is in fact non-conformant, a *Rapide* Map is defined that creates an equivalence to events of the Reference model Job_Assignor.

```
MAP m () FROM JA_B:Job_Assignor_Interface_App to
        Job_Assignor_Interface_RCS IS
RULE
```

```
#1
(?tcf : String)
        JA_B.Do_Task_Bad(?tcf)
        ||>
        Do_Task(?tcf);; –

#2
(?tcf, ?status : String)
        JA_B.Schedule_Job_Bad (?tcf) ->
        JA_B.JA_Status_Bad(?status)
        ||>
        Schedule_Job(?tcf) ->
        JA_Status(?status);;

END;
```

This segment shows the mapping of the two sets of independent events in the Job_Assignor application–symbolized by "||>"–onto their equivalents events in the Reference Model specification. The first set consists only of Do_Task; the second contains Schedule_Job and Status. This mapping is illustrated graphically in POSET notation in Figure 4.4.

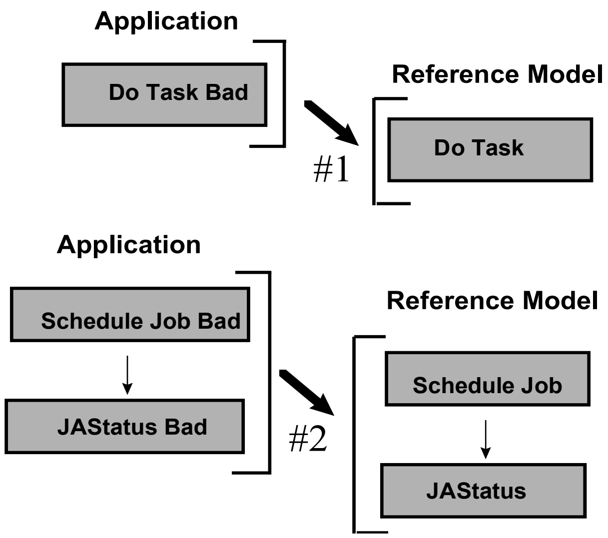

Figure 4.4: Event Mappings from Application to Reference Model specification

Execution of the Specification to show non-conformance. The Reference Model specification was then executed with mapped events from the Job_Assignor specification. The result is shown in Figure 4.5.

The POSET in Figure 4.5 replicates the POSET of the non-conforming Job_-Assignor. In addition, two error message events are generated. One represents

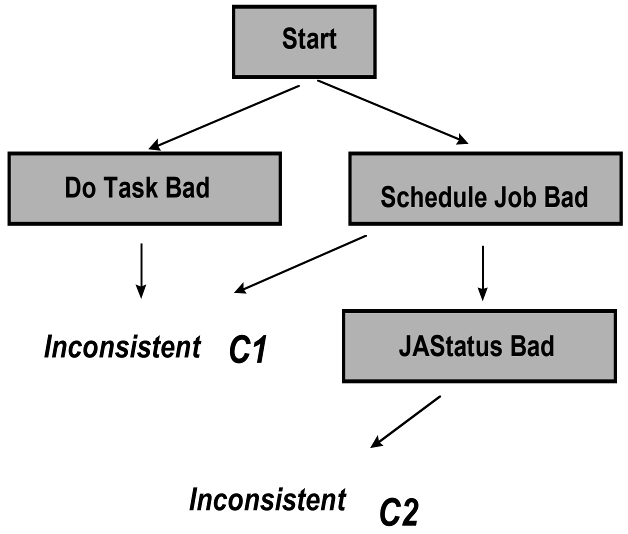

Figure 4.5: Result of execution of mapped events

the violation of constraint C1 on the Reference Model Job_Assignor, triggered by the causal independence of the Do_Task and Schedule_Job Events. The second violates constraint C2 since Do_Task and Status events are independent.

This capability aroused considerable interest among 4-D/RCS experts who are interested in a means of establishing conformance of application systems to the Reference Model Architecture. However as demonstrated above, in *Rapide* the Map facility currently is limited to mapping of events associated with behavior. 4-D/RCS domain experts maintain there is greater benefit in demonstrating conformance of structure or system design. This kind of verification would include existence of particular modules, specific messages, and connectivity of modules so that specific messages are exchanged between modules. Work on extending *Rapide* to demonstrate conformance to system structure is reported in (Vera et al., 1998).

Chapter 5

Conclusions

General conclusions are first provided on the use of ADLs for 4-D/RCS. This is followed by specific conclusions of two kinds: (1) using ADLs for communicating RCS architectural structure and system behavior; and (2) using ADLs to provide a formal basis for developing automated software support tools to check specifications, including tools for verifying conformance of an application system design against the Reference Model Architecture. Conclusions of the first kind take into consideration direct interactions between the user and the ADL while conclusions of the second kind may not. Conclusions are provided about the use of ADLs for representing the 4-D/RCS Reference Model Architecture and about the potential for using ADLs to define architectures and software components for specific 4-D/RCS applications. The potential of ADLs in facilitating automated reuse of software architectures and component specifications is discussed. These conclusions provide a basis for the ADL requirements and research directions.

5.1 General Conclusions

The *Rapide* ADL proved to be a viable tool for formalizing 4-D/RCS structure and behavior, though a number of specific improvements are recommended. To date, it is the most rigorous representation of the Reference Model Architecture. The specification also demonstrated that ADLs were potentially useful for extending and refining the 4-D/RCS Reference Model Architecture. The transfer of ADL concepts, such as structure, components, interconnectivity, and analysis into other software development tools should provide benefits to software technology in general. The focus of ADLs is on the design phase of the software development process. Automated Component-based software development cannot be fully realized until ADLs are integrated into a more comprehensive methodology with other phases including system definition, analysis, and implementation (Senehi and Kramer 1998; SPC 1992; STARS 1993).

5.1.1 Use of ADLs to Specify and Analyze 4-D/RCS

Based on informal review by 4-D/RCS experts, the Intelligent Control Node specification was successful in capturing and representing major RCS architectural concepts. There were no concepts that could not be represented. However, the specification had to be simplified, modified to allow the application of specific RCS keywords, and supplemented by the use of graphical support. Examination of other ADLs that have the same language features led to the conclusion that some 4-D/RCS structural concepts could also be represented in other ADLs including, but not limited to, Aesop (Melton, 1998), SADL (Moriconi and Riemenschneider, 1997), UniCon (Zelesznik, 1996), and Wright (Allen, 1997).

The simulated execution of the Control Node Architecture reinforced the specification and proved to be a valuable aid in communicating the architecture by enabling reviewers to visualize the topology and high-level execution of the Intelligent Control Node. The successful representation of system behavior, though limited by the amount of resources for this study, depended upon support by the ADL for the 4-D/RCS computational model. Automated analysis functions based on formal methods approaches are much needed, but more research is necessary in this area.

As a long-term goal, ADLs should allow specifications to be stated at a sufficiently high level of abstraction for non-computer scientists. Very abstract architecture specifications should be possible that are easier to understand than a program written in Fortran or Pascal. Ultimately, the existence of a domain-specific ADL for 4D/RCS would provide significant advantage.

5.1.2 Using ADLs to Further Develop the 4-D/RCS

The ability to create a precise, communicable specification of the 4-D/RCS Reference Model Architecture led to potential improvements to the architecture itself. Two possible changes to the Architecture as described in (Albus and Lumia,1994) were identified:

1. In the model described in Chapter 4, Job_Assignor applies a Fetch_Task_frame operation to retrieve the task knowledge necessary for task decomposition. Although this operation is not explicitly stated in the 4-D/RCS Reference Model, we found it consistent with the usage of task frames and found it effective in our experiment. Therefore, this operation may be proposed as one of the accepted Job_Assignor functions in its specification.

2. A prototype set of exception and error handling messages was defined for the modules in the Control Node specification. The flow of these messages was identified to create an error-handling "sub-architecture," in the *Rapide* context, whose operation may be simulated using the *Rapide* support tools. The message set, after being fully developed and tested in *Rapide*, may be proposed for inclusion in the Reference Model Architecture.

These illustrate the potential of ADLs as practical tools for development of the Reference Model Architecture and software designs in general.

5.1.3 Transfer of ADL Concepts into Other Real-Time Development Support Tools

Presently, there are a number of public domain and commercially available software support tools for design and simulation of real-time software systems. These tools have well-developed facilities for designing and implementing individual software systems. However, they do not typically provide any guidance to users about how to structure their system or make other design decisions. ADLs introduce notions of software architecture that could potentially provide additional structure in order to improve the capabilities of these tools. Users or enterprises could set preferences in term of which architecture or architectural style is to be used in developing systems. The tools would then either guide designers as the system is being developed or could flag situations where the architecture or style are violated. For companies interested in building a system that conforms to a given architecture, this type of support in a tool would be extremely valuable in ensuring conformance throughout development. Further effort is necessary to explore the potential of infusing ADL concepts into real-time development support tools. This avenue could provide the benefits of ADLs to end users while shielding them from having to learn a new language and concepts. The real-time development tools would guide users in constructing systems per rules for a prescribed architecture through their graphical user interfaces. The users would not be burdened with the underlying mechanics of the ADL specification. In addition to design, analysis and simulation capabilities from the ADLs could be incorporated into the tools. The tools could generate executable or source code. This would automatically assure traceability from the desired architecture through to the executable code. Eventually, tools using ADLs could support highly-automated composition of real-time systems from existing or tailorable components.

5.2 Specifying 4-D/RCS System Structure and Behavior

In this section, conclusions derived from the prototype specification are followed by additional requirements that must be considered to use ADLs to specify 4-D/RCS software architectures and components.

5.2.1 Specifying Structure of 4-D/RCS Reference Model Architecture

Hierarchical Architectural Style. Aided by the simplification of the specification and the use of graphics, the Control Node module Interfaces and signatures were clearly defined. Module connections, even though not definable in *Rapide* as explicit types–or first-class objects–were also easily communicated. The successful representation of the 4-D/RCS Control Node hierarchy indicates that representation of other parts of the seven-level architecture described in Chapter 2 should be possible. The ability to communicate 4-D/RCS architecture would be improved by defining a specific *architectural style* for top-down hierarchical structures. The definition of *architectural styles* for real-time control system software would further benefit from studies in control system software architecture frameworks (Senehi and Kramer, 1998).

Domain-Specific Syntax. The syntactic description was simplified and altered to conform to the descriptive forms familiar to 4-D/RCS experts. 4-D/RCS experts found specifications much easier to understand when RCS terminology was used. As an example of this approach, instead of declaring an RCS module such as SCHEDULER as a *component* or *interface type*, it should be possible to introduce a higher-level language type called *RCS_Module* in a specification. Once defined, *RCS_Module* could serve as a "meta type" for the definition of interface types that are specific to 4-D/RCS such as SCHEDULER. This argues for the development of either a flexible ADL with an extensible syntax that can be specialized for 4-D/RCS or a domain-specific ADL that utilizes RCS terminology.

5.2.2 Research Directions in Representing Hierarchical 4-D/RCS Architectures

Owing to evidence in biological systems and theory of control science, RCS prescribes rules for decomposing the control hierarchy for a system. In his "Outline for a Theory of Intelligence," (Albus, 1991), Albus proposed that:

"In a hierarchically structured, goal-driven sensory interactive, intelligent control system architecture:

1. control bandwidth decreases about an order of magnitude at each higher level,
2. perceptual resolution of spatial and temporal patterns decreases about an order of magnitude at each higher level,
3. goals expand in scope and planning horizons expand in space and time about an order of magnitude at each higher level, and
4. models of the world and memories of events decrease in resolution and expand in spatial and temporal range by about an order of magnitude at each higher level."

These English language rules must be encoded into ADLs in order to represent fully the semantics of an RCS system. Temporal scales and spatial extents relative to other levels of the hierarchy must be represented and validated. While existing ADLs can meet some of these requirements, further work on ADLs adding methods to define these measures and to express constraints among them is needed to allow specifications such as those quoted to be stated and applied.

5.2.3 Specifying System Behavior for 4-D/RCS Systems

Communication of behavior description for components of the 4-D/RCS prototype specification proved more difficult than communication of component structure. First, specifying behavior in ADLs involves use of a larger, more complex set of language primitives than is needed for specifying interface signatures, connections, and architectures. Second, specifying behavior normally requires understanding the underlying computational model upon which the language is based. Finally, system behavior in 4-D/RCS has very significant complexity, being based on theory from artificial intelligence, control systems, and other disciplines. Therefore, the effort needed in both learning more complex behavior description language and comprehending complex specifications is potentially greater.

To facilitate communication of 4-D/RCS system behavior, an ADL must provide an effective means for abstractly specifying algorithms, component behavior, and performance. While some ADLs may allow representation of all or most of the behavior needed for 4-D/RCS, this requirement may lead to defining additional language constructs to more directly represent specific 4-D/RCS behavior. It may also require additional facilities for guiding developers in generating their component specifications, through for example, templates that they can fill in, as proposed in (Horst et al, 1997) and (Messina et al., 1999). As with system structure, such capabilities would allow ADLs to specify essential aspects of behavior at a higher level of abstraction than for programming languages. These capabilities could be part of a domain-specific ADL with a syntax that is customized for 4-D/RCS systems.

Finite-State Machine (FSM). The predominant computational model for describing behavior in 4-D/RCS is the FSM. Experience gained from many years of building 4-D/RCS systems led to the conclusion that, while any computing language with *if..then..else* constructs can express FSM concepts, greater advantage is gained with a language that provides FSM-specific constructs such as state-graphs. Examining language documentation led to the conclusion that the *Rapide* POSET computational model can support specification of FSM behavior (as can a number of other ADLs). For instance, state graphs can automate the specification and hide the housekeeping details of states, transitions, stimuli, and constraints on behavior. The development of ADL language primitives for supporting specification of FSM-oriented behavior is preferable and provides a basis for the possible development of a domain-specific ADL for 4-D/RCS.

Artificial Intelligence Programming Techniques. For certain applications, 4-D/RCS planning functions require use of artificial intelligence search methods. To represent system behavior in 4-D/RCS architectures for such applications, an ADL should therefore facilitate specification of the high-level behaviors or performance characteristics for processing algorithms using search methods such as depth-first, breadth-first, and others. For example, real-time systems may benefit from depth-first search, since the system will be more likely to have a complete solution to act on, albeit suboptimal, if required by timing constraints. Similarly, sensing subsystems will benefit from obtaining a suboptimal solution quickly, improving the solution as time permits. These performance profiles should be part of the system simulation in an ADL.

Control Theories and Hybrid System Constructs. In addition, an ADL should support the ability to specify feedback and control theory and discrete time and event theory. From the control system theory point of view, RCS possesses a *hybrid system* construct, meaning that RCS based systems utilize both continuous-time based and discrete-event based algorithms.

Other Problem Solving Paradigms. Within 4-D/RCS application systems, a wide variety of other types of algorithms are necessary for specific functions involving intelligent planning, sensory processing, and value judgement. Much of this functionality is application-specific, meaning that different algorithms are necessary to accomplish a similar task in different domains such as, for instance, autonomous vehicle control and manufacturing. To represent architectures for such systems, it is desirable that an ADL based application system model could link in these algorithms so that effective tests can be performed.

5.2.4 Other RCS Requirements

Additional RCS requirements need to be considered for real-time system processing, parallel processing, and general, infrastructural types of tasks. While not requirements for the 4-D/RCS Reference Model Architecture, these capabilities are needed to provide complete definition of specific system designs. Some existing ADLs including *Rapide* provide these capabilities. However, these requirements could provide a basis for developing language constructs for a domain-specific ADL for RCS with a syntax that is specific to RCS and can be used to define RCS software architectures and components.

Additional requirements for real-time programming. Since 4-D/RCS is for designing real-time systems, an ADL should define notions of

1. duration in time of processes,
2. mixed asynchronous and synchronous processing,
3. spatial scope of a process or set of processes,
4. algorithm and component complexity, and
5. determinism in execution.

Requirements for serial and parallel processing. It is often important to be able to divide the processing into atomic processing components (versus a single monolithic component) that can be executed serially or in parallel which will facilitate process cessation and make it deterministic. Therefore, it is important that an ADL be able to specify these processing characteristics, namely, process cessation, process modularization, parallel and serial execution of atomic processing components interlaced with other atomic processing components from other processes. Serial and parallel processing capabilities are provided by some ADLs, including *Rapide*.

Infrastructural Requirements–"Housekeeping". RCS implementations contain generic "housekeeping" types of actions. These include checking a module's input command, reading in the current world model, processing the current command, and writing out the updated world model. Following a convention, all of these messages carry identification numbers that are to be matched between the issuers and the responders. These can be readily captured in *Rapide*. Because these behaviors were part of the implementation mechanics, the authors did not attempt to represent them in the study.

Infrastructural Requirements–Performance. It is desirable to allow capturing performance statistics, such as timing, states, and errors. These would be useful in system diagnostics and maintenance. It should be noted that *Rapide* does provide the capability to capture time-related data which could not be exercised in this study due to resource limitations.

Infrastructural Requirements–Human Interface. RCS requires that operators be able to interact with the control systems with various degrees of involvement, ranging from monitoring the functioning of a subsystem to teleoperation of a device or vehicle. This function may be implemented with text or graphics, using whatever mechanism is appropriate, such as hand-held control devices or Internet-based remote access. It is desirable that an ADL allow specifying this function at a high level. For example, if an operator acknowledgement of an alarm condition is required when certain events are triggered, there should be some basic representation of this interaction. It is unclear whether it is necessary for an ADL to support a higher-fidelity simulation of the device in order to exercise the Operator Interface functionality.

5.3 ADLs and Software Development Support Tools

Conclusions are provided on the relationship between ADLs and software support tools. These conclusions are based not only on the study but review of existing ADL products.

5.3.1 Tool Support for Analysis of Specifications

One of the benefits of rigorously specifying 4-D/RCS designs is that it is possible to check the completeness and internal consistency of the reference model architecture before it is used as a basis to develop individual system designs. By providing a basis for formalized, or at least rigorous specification, most ADL products surveyed also provide a basis for development of automated analysis capabilities. In the case of *Rapide*, analysis is based on simulation of the execution of a system architecture and analysis of POSET traces. This proved to be valuable for visualizing, understanding, and verifying system behavior.

Other ADLs take different approaches using automated tools for analysis of specifications based on formal methods approaches. *SADL* (Moriconi and Riemenschneider, 1997) uses w-logic, a weak second-order logic, as a basis for proving the correctness of mappings between architectures at different levels of abstraction. *Wright* (Allen, 1997) uses First-Order Logic to specify constraints and a Communicating Sequential Processes (CSP) computational model to specify behavior of components and connections, providing a basis for a set of automated checks on specification consistency and completeness. Examples from *Wright* are checks that determine the existence of a deadlock condition within the specification of the behavior of an architecture and checks to determine compatibility between connections and components (called part/role compatibility). Ideally, formal methods approaches and simulation should supplement and complement each other. Further investigation is necessary to identify and catalog checks supported by existing ADLs that are either under consideration as being useful or have been demonstrated as being useful.

5.3.2 Verification of Designs of RCS Systems Against the Standardized Reference Model Architecture

The use of an ADL to verify the behavior of an application system design against the Reference Model Architecture is demonstrated as a proof-of-concept in the Control Node prototype. However, 4-D/RCS domain experts maintain that verification of the system topology is at least equally important for the Reference Model Architecture. This form of verification involves showing that the application system contains the same basic structure including components, event connections, and data structures as the Reference Model. As a result, two kinds of verification are important from the standpoint of 4-D/RCS:

1. Verification to the structure of the Reference Architecture including existence of specific components, events, and control flows.

2. Verification of behavior, including behavior within components and behavior across component connections and an entire architecture.

Verification of behavior is the focus of *Rapide*. Further research is necessary to define techniques for demonstrating consistency with system topology. As indicated earlier, work in extending *Rapide*'s POSET model to verification of system structure has been reported in (Vera et al., 1998). In SADL, (Moriconi et al., 1995) describes a general approach, called *architectural refinement*, that utilizes theorem proving techniques. In this approach, proofs are constructed to show that in the case when a more general or abstract architecture is applied to produce a more detailed design, that any system that correctly implements the more detailed design also correctly implements the abstract architecture. Refinement is used to demonstrate correctness with respect to the connectivity of events between modules at different levels of abstraction; and this approach may be applicable to the problem of verifying application system designs. As with verification of internal consistency, further work is needed on the use of theorem proving techniques on this problem.

It is possible that significant advantages could be provided by the use of *architectural styles* to constrain architectures to specific topologies. Application systems could be tested to determine if they conform to the constraints specified in a particular style. The work of (Moriconi et al., 1995), in part, does consider the use of styles. Despite the need for additional research on the use of ADLs for application system verification, ADLs do provide a viable conceptual basis for automated verification of system designs against a canonical architecture needed for RCS.

5.4 ADLs and Component-Based Software Reuse

Though this limited study did not have the resources to explore reuse possibilities, there is potentially a strong relationship between ADLs and automated component-based software reuse. Having an unambiguous definition of the RCS architecture provides a framework for reuse as well. In a scenario where several organizations collaborate on the implementation of an RCS-based system, not only are they concerned with conformance to the reference architecture, but also with communicating how the various components fit together and behave together. An ADL can prove valuable in this function. ADLs go beyond providing just the signature specification for a component or subsystem. They allow developers to see the big picture and where their particular pieces fit in and how the pieces are expected to behave or interact with the rest of the system. The feasibility of applying ADLs to facilitate reuse was somewhat limited by the developers' abilities to comprehend the specification in a given ADL and how to interpret it for their particular implementation job. If developers can overcome the initial challenges of becoming familiar with the ADL's notation and conventions, they may find them a valuable part of a reuse strategy. The development of domain-specific ADLs can be expected to be helpful in this regard as well. Simulation of components provides additional benefits not available in typical notations or descriptions of software components.

<u>Extending ADLs to Support Software Reuse</u>. ADLs could be extended to support reuse with additions of specific language features based on reuse concepts from the literature on domain engineering (Kang et al., 1990; SPC, 1992; STARS, 1993), thus providing a basis for automation of software development. Domain engineering is the process of developing reusable software for a family of systems with similar requirements. One addition would be to make explicit within a specification those parts that are invariant and those parts that vary and can be adapted for individual systems designs. This is accompanied by guidance to developers on how to modify and adapt the variable parts for reuse. For instance, in the intelligent control node architecture, the Job_Assignor component may always be required to be present in all 4-D/RCS systems. Certain events generated by the Job_Assignor may be invariant such as the Do_Task

event. However, the type and number of parameters passed may need to be varied depending on the application. Other events may be defined as being optional. In addition, an architecture specification may identify optional components, parameterizable components, or even entire subarchitectures that can be varied. Guidelines would be used by developers with the aid of support tools to select options and customize the specification for particular applications. This concept could be further extended by the use of software support tools that assist developers in selecting and modifying system designs and components. The resulting system specifications potentially could be automatically composed and generated using the support tools. An example of such a system for automated generation of system requirements is provided in (Dabrowski and Watkins, 1994). The ample literature on research into domain engineering methods provides a resource for identifying additional possibilities in this area. This research together with the ongoing work on ADLs provides an important basis for automation of software development.

Integrating ADLs into the Domain Engineering Process. Research in domain engineering has resulted in the creation of methodologies that provide a comprehensive approach to development of reusable components that encompass all phases of the software lifecycle (SPC, 1992; STARS, 1993). The use of this approach for development of control system software architectures has been advocated in (Senehi and Kramer, 1998). Domain engineering is based on the assumption that the development of reusable software components requires domain definition and analysis phases in addition to development of architectures and components for domains. In the domain definition phase, the scope of a family of systems is defined; in the domain analysis phase, software requirements for the domain are set forth. Domain engineering provides languages for defining domains and domain software requirements. To use ADLs in a domain engineering process, it will be necessary to have some level of integration between ADLs and languages for describing domain software requirements. Similarly, it is important to have links between reusable requirements specifications and software architecture specifications in order to define a clear component reuse process and establish traceability. A consistent domain engineering process that includes ADLs is one possible approach to realizing the potential of ADLs for automated reuse of software components.

Domain-Specific ADLs and Software Reuse. ADLs can be significantly enhanced through the development of domain-specific syntax for abstractly describing structure and behavior in software architecture and components and adding language features for supporting reuse. Basing ADL semantics on formalized theories of architecture and system behavior allows definition of analytical functions that can automatically determine internal consistency of reference architectures, software system designs, and individual components. The combination of these facilities with graphical software support tools would result in powerful tools for automated component-based software engineering. It remains a subject for future research as to whether this potential will be realized in ADLs, will be incorporated into existing commercial development support tools, or will emerge as a new genre of software technology.

This report has provided the results of an investigation into the use of architectural description languages to represent the RCS Reference Model Architecture and RCS software components. ADLs have the capabilities to represent RCS and to be useful tools for further developing RCS. However, several areas of research are suggested in order to make ADLs more effective tools for RCS software specifications. These include creation of a domain-specific syntax for RCS, language features for describing behavior in terms of RCS computational models, language features for verification of adherence to RCS Reference Model structure, and support for software reuse. Transfer of ADL concepts into existing real-time software development tools is another important direction to pursue. It is the hope of the authors that this work provides a contribution towards both the development of ADLs as tools for software component technology and the formalization of the 4-D/RCS Reference Model Architecture.

References

Albus, J.S. and Blidberg, D.R. 1987. Control System Architecture for Multiple Autonomous Undersea Vehicles (MAUV). Proc. of the Fifth International Symposium on Unmanned, Untethered Submersible Technology, Merrimack, NH.

Albus, J.S., Lumia, R., Fiala, J., and Wavering, A. 1989. NASREM - The NASA/NBS Standard Reference Model for Telerobot Control System Architecture. Proc. of the 20th International Symposium on Industrial Robots, Tokyo, Japan.

Albus, J. S. 1991. "Outline for a Theory of Intelligence. IEEE Transactions on Systems, Man, and Cybernetics, Vol. 21, No. 3:473-509.

Albus, J.S., Lumia, R. 1994. The Enhanced Machine Controller (EMC): An Open Architecture Controller for Machine Tools. Journal of Manufacturing Review, Vol. 7, No. 3, pgs. 278-280.

Albus, J. S. 1995. The NIST Real-time Control System (RCS): An Application Survey. Proc. of the AAAI 1995 Spring Symposium Series, Stanford University, Menlo Park, CA.

Albus, J. S., and Meystel, A. 1996. A Reference Model Architecture for Design and Implementation of Intelligent Control in Large and Complex Systems. International Journal of Intelligent Control and Systems, Vol. 1, No. 1, pp. 15-30.

Albus, J. S. 1997. 4-D/RCS: A Reference Model Architecture for Demo III. National Institute of Standards and Technology, Gaithersburg, MD, NISTIR 5994.

Allen, R. 1997. A Formal Approach to Software Architecture. PhD Thesis, Carnegie Mellon University, Pittsburgh, PA, Technical Report Number: CMU-CS-97-144.

Barbera, A. J., Albus J. S., Fitzgerald M.L., and Haynes L.S. 1984. RCS:The NBS Real-Time Control System. Proc. of Robots 8 Conference and Exposition. Detroit, MI, pp. 1-19.

Dabrowski, C. and Watkins, C. 1994. A Domain Analysis of the Alarm Surveillance Domain. National Institute of Standards and Technology, Gaithersburg, MD, NISTIR 5494.

Dickmanns, E. D. et al. 1994. The Seeing Passenger Car "VaMoRS-P," Proc. International Symposium on Intelligent Vehicles 'I94, Paris, France, pp.68-73.

Feijs, L.M.G. and Jonkers, H.B.M. 1992. Formal Specification and Design, Cambridge University Press, Victoria, Australia.

Garlan, D., and Shaw, M. 1994. Characteristics of Higher-Level Languages for Software Architecture. Software Engineering Institute, Carnegie-Mellon University, Pittsburgh, PA, CMU/SEI-94-TR-23.

Garlan, D., and Perry, D. 1995. Introduction to the Special Issue on Software Architecture. IEEE Transactions on Software Engineering, Vol. 21, No. 4, pp. 269-274.

Horst, J. A. 1993. Coal Extraction Using RCS. Proc. of the 8th IEEE International Symposium on Intelligent Control, Chicago, IL, pp. 207-212.

Horst, J. A., Messina, E., Kramer, T., Huang, H. M. 1997. Precise Definition of Software Component Specifications. Proc. of the 7th Symposium on Computer-Aided Control System Design (CACSD '97), Gent, Belgium, pp.145-150.

Huang, H. and Messina, E. 1996. NIST-RCS and Object-Oriented Methodologies of Software Engineering: A Conceptual Comparison. Proc. of the Intelligent Systems: A Semiotic Perspective Conference, Vol. 2: Applied Semiotics. Gaithersburg, MD, pp. 109-115.

Huang, H.M., Scott, H., Messina, E., Juberts, M., Quintero, R. 1999. Intelligent System Control: A Unified Approach and Applications, Chapter in Gordon and Breach International Series in Engineering, Technology and Applied Science, Volumes on "Expert Systems Techniques and Applications," To be published in 1999.

Kang, K., Cohen S. , Hess J., Novak W., and Peterson S. 1990. Feature-Oriented Domain Analysis (FODA) Feasibility Study. Software Engineering Institute, Carnegie-Mellon University, Pittsburgh, PA, CMU/SEI-90-TR-21.

Luckham, D. 1996. Rapide: A Language and Toolset for Simulation of Distributed Systems by Partial Ordering of Events. Stanford University, Palo Alto, CA. CSL-TR-96-705.

Medvidovic, N. and Taylor R. 1999. Classification andComparison Framework for Software Architecture Description Languages. Accepted for publication in IEEE Transactions on Software Engineering.

Melton, R. 1998. The Aesop System: A Tutorial. Carnegie Mellon University, Pittsburgh, Pennsylvania.

Messina, E., Horst, J., Kramer, T., Huang, H. Michaloski, J. 1999. Component Specifications for Robotics Integration. To appear in Autonomous Robots Journal, Volume 6, No. 3.

Moriconi, M., Qian, X. and Riemenschneider, R. 1995. "Correct Architecture Refinement. IEEE Transactions on Software Engineering, Volume 21, Number 4, pp.356-372.

Moriconi, M and Riemenschneider, R. 1997. Introduction to SADL 1.0: A Language for Specifying Software Architecture Hierarchies. Stanford Research Institute, Palo Alto, CA, TR SRI-CSL-97-01.

OMG. 1999. RFP: UML Profile for Scheduling Performance, and Time Object Management Group Document ad/99-03-13. Object Management Group, Framingham, MA. http://www.omg.org.

Proctor, F. M. and Shackleford, W. 1999. http://isd.cme.nist.gov/projects/rcs_lib/.

Senehi, M. and Kramer, T. 1998. A Framework for Control Architectures. International Journal of Computer Integrated Manufacturing, Volume 11, Number 4, pp. 347-363.

Shackleford, W., Proctor, F.M. 1998. JAVA-Based Tools for Development and Diagnosis of Real-Time Control Systems. Proc. of the ASME: Computers in Engineering Conference. Atlanta, GA.

Shaw, M. 1994. Comparing Architectural Design Styles. IEEE Software, November, 1994, pp. 27-41.

Shoemaker, C. M. and Bornstein, J. A. 1998. Overview of the Demo III UGV program. Proc. of the SPIE Robotic and Semi-Robotic Ground Vehicle Technology , Vol. 3366, pp.202-211.

SPC 1992. Domain Engineering Guidebook, Software Productivity Consortium. Herndon, VA. SPC-92019-CMC, Version 01.00.03.

STARS. 1993. Organizational Domain Modeling, Volume I - Conceptual Foundations, Process And Workproduct Description, Informal Technical Report for the Software Technology for Adaptable, Reliable Systems (STARS), Report Number STARS-UC-05156/024/00.

USPS. 1991. Stamp Distribution Network, Advanced Technology & Research Corporation, Burtonsville, MD. USPS Contract Number 104230-91-C-3127 Final Report.

Vera, J., Perrochon, L., Luckham, D. 1998. Event-Based Execution Architectures for Dynamic Software Systems. Proc. TC2 First Working IFIP Conference on Software Architecture (WICSA1). San Antonio, Texas, USA. Kluwer. pp. 303-317.

Vestal, S. 1993. A Cursory Overview and Comparison of Four Architecture Description Languages. Honeywell Technology Center, February 1993.

Zelesnik, G. 1996. The UniCon Language Reference Manual. Carnegie Mellon University, Pittsburgh, Pennsylvania. http://www.cs.cmu.edu:80/afs/cs.cmu.edu/project/vit/www/unicon/reference-manual/Reference_Manual_1.html.

Appendix A: The *Rapide* Specification

This appendix contains a draft *Rapide* specification of the software architecture, module interfaces, and module connections for a single 4D/RCS Intelligent Control Node. The listing below provides a guide to the organization and content of this specification. Please note that this specification excludes those parts of the *Rapide* program that are necessary for animation of a sample execution of the architecture. Also, the parts of the specification are reordered for purposes of presentation.

Organization of *Rapide* Specification

A.5 Global Declarations

A.5.1 Global Variables

```
-- ****************************************************
-- **                                                **
-- **     THESE ARE VARIABLES USED IN                **
-- **     RCS CONTROL NODE AND ITS COMPONENT MODULES **
-- **                                                **
-- ****************************************************


TYPE  Plan IS string;
TYPE  Schedule IS string;
TYPE  result IS string;

-- ****************************************************
-- Declare resources controlled by control nodes
-- at next lowest level in the RCS hierarchy.
-- ****************************************************
--
Num_Controlled_Resources : var integer := 3;
resource : array[integer] of Controlled_Resources
           IS (1..3, "Comp1", "Comp2", "Comp3");
```

A.5.2 Global Complex Data Structures

```
-- **********************************************************************
--  Declare TASK_COMMAND_FRAME as string variable for now.
--  Declare record structure for TASK_FRAME.  This is highest
--  level abstract type for Task Frame from which subtypes may be
--  declared.  Array of these record structures is indexed by TASK_NAME
-- **********************************************************************

TYPE Task_Command_Frame IS STRING;

TYPE Task_Frame IS RECORD
  task_name       : ref(string);
  task_process    : ref(plan);
END; -- record

TYPE Move_Task IS RECORD
  INCLUDE Task_Frame;
```

```
  Coordinates       : ref(string);
END; -- record
```

A.6 4D/RCS Control Node

A.6.1 Interface for RCS Control Node

```
TYPE RCS_Node_Interface IS INTERFACE
ACTION
IN
     Do_task (j : Task_Command_Frame),
     RCV_SP_Data (Obj : string; Data : string),
     RCV_SubNode_Status (ST : string),
     RCV_Request_KB_Object (Obj : string),
     RCV_KB_Object (Obj : string),
     Operator_Input (Inp : string);
OUT
     SND_SP_Data (Obj : string; Data : string),
     Do_sub_task (CR : Controlled_Resources; J : Task_Command_Frame),
     FWD_Request_KB_Object (Obj : string),
     SND_KB_Object (Obj : string),
     Status (CR : Controlled_Resources; ST : string),
     Operator_Output (Outp : string);

END
```

A.6.2 *Rapide* Architecture for RCS Node

```
ARCHITECTURE RCS_Node_Architecture () RETURN RCS_Node_Interface IS

   BG : Behavior_Generator_Interface IS BG_Module_Architecture();
   WM : World_Modeling_Interface IS World_Modeling_Architecture();
   VJ : Value_Judgement_Interface IS Value_Judgement_Module();
   SP : Sensory_Processing_Interface IS Sensory_Processing_Module();

CONNECT
   (?J : Task_Command_Frame)
   Do_task (?J) ||> BG.Do_task(?J);

   (?CR : Controlled_Resources; ?ST : string)
   BG.BG_Status (?ST) ||>   Status ("Node", ?ST);
```

```
   (?CR : Controlled_Resources; ?J : Task_Command_Frame; ?S : Schedule)
   WM.FWD_REQ_Evaluate_Schedule (?CR, ?J, ?S)  ||>
       VJ.RCV_Evaluate_Schedule (?CR, ?J, ?S);

-- ** Connections Between Value Judgement and  **
-- ** Behavior_Generation  Interfaces          **

   (?CR : Controlled_Resources; ?J : Task_Command_Frame;
    ?S : Schedule; ?RS : Result)
   VJ.SND_Schedule_Evaluation (?CR, ?J, ?S, ?RS) ||>
       BG.RCV_Schedule_Evaluation (?CR, ?J, ?S, ?RS);

   (?CR : Controlled_Resources; ?J : Task_Command_Frame;
    ?S : Schedule; ?ST : String)
   VJ.SND_VJ_Status (?CR, ?J, ?S, ?ST) ||>
   BG.RCV_VJ_Status (?CR, ?J, ?S, ?ST);


-- ** Connections Between World Modeling and  **
-- ** Behavior_Generation Interfaces          **

  (?J : Task_Command_Frame)
   BG.FWD_Fetch_task_frame (?J) ||>
       WM.RCV_Fetch_task_frame(?J);

  (?J : Task_Command_Frame; ?TF : Task_Frame)
   WM.FWD_task_frame (?J, ?TF) ||>
      BG.RCV_task_frame (?J, ?TF);

  (?CR : Controlled_Resources; ?J : Task_Command_Frame; ?S : Schedule)
   BG.FWD_Post_Schedule (?CR, ?J, ?S) ||>
       WM.RCV_Post_Schedule (?CR, ?J, ?S);

   (?CR : Controlled_Resources; ?J : Task_Command_Frame; ?S : Schedule)
   BG.FWD_Simulate_Schedule (?CR, ?J, ?S) ||>
       WM.RCV_Simulate_Schedule (?CR, ?J, ?S);

   (?CR : Controlled_Resources; ?J : Task_Command_Frame; ?S : Schedule)
    WM.FWD_Simulation_Failure_Notification (?CR, ?J, ?S) ||>
       BG.RCV_Simulation_Failure_Notification (?CR, ?J, ?S);

-- ** Connections Between External Device and  **
-- ** Sensory Processing Module                **

--     (?Data : string)
```

```
--    Sensor_Output (?Data) ||>
--        SP.SP_RCV_Observed_Input (?Data);

-- ** Connections Between Sensory Processing   **
-- ** Modules at different levels of RCS       **

    (?Obj : string; ?Data : string)
     SP.SP_SND_Update (?Obj, ?Data) ||>
        SND_SP_Data (?Obj, ?Data);

   (?Obj : string; ?Data : string)
      RCV_SP_Data (?Obj, ?Data) ||>
        SP.SP_RCV_SP_Data (?Obj, ?Data);


-- ** Internal Connections Between Sensory Processing **
-- ** and World Modeling & Value Judgement            **

    (?Obj : string; ?Data :string)
    SP.SP_SND_Update (?Obj, ?Data) ||>
       VJ.RCV_Update (?Obj, ?Data);

    (?Obj : string; ?Data :string)
    SP.SP_SND_Update (?Obj, ?Data) ||>
       WM.RCV_Update (?Obj, ?Data);

    (?CR : Controlled_Resources; ?J : Task_Command_Frame)
    BG.FWD_Do_Sub_Task (?CR, ?J) ||>
      Do_Sub_Task (?CR, ?J);

END;

--
-- END of RCS Node Declarations
```

A.7 RCS Node Submodules

```
-- ************************************************************
--
--   DECLARATION OF BEHAVIOR GENERATION, WORLD MODELING
--   AND VALUE JUDGEMENT MODULES
--
-- ************************************************************
```

A.7.1 Behavior Generation

A.7.1.1 Interface for Behavior Generation Module

```
TYPE Behavior_Generator_Interface IS INTERFACE;
ACTION
  IN
     Do_task (J : Task_Command_Frame),
     RCV_task_frame (J : Task_Command_Frame; TF : Task_Frame),
     RCV_Schedule_Evaluation (CR : Controlled_Resources;
                              J : Task_Command_Frame;
                              S : Schedule; RS : Result),
     RCV_Simulation_Failure_Notification (CR : Controlled_Resources;
                                          J : Task_Command_Frame;
                        S : Schedule),
     RCV_VJ_Status (CR : Controlled_Resources;
                                J : Task_Command_Frame;
                                S : Schedule;
                                ST : String);
OUT
     FWD_Fetch_task_frame (J : Task_Command_Frame),
     FWD_Simulate_Schedule (CR : Controlled_Resources;
                            J : Task_Command_Frame; S : Schedule),
     FWD_Post_Schedule (CR : Controlled_Resources;
                        J : Task_Command_Frame; S : Schedule),
     FWD_Do_Sub_Task (CR : Controlled_Resources;
      J : Task_Command_Frame),
     BG_Status (ST : String);

CONSTRAINT
 --
 --     Do not allow Do Task and Do_Sub_Task events
 --     to be causally independent.
 --
 NEVER
    (?J : Task_Command_Frame; ?CR : Controlled_Resources)
    Do_task (?J) || FWD_Do_Sub_Task (?CR, ?J);

END;
```

A.7.1.2 Architecture for Behavior Generation Module

```
ARCHITECTURE BG_Module_Architecture ()
RETURN Behavior_Generator_Interface IS

   JA  : Job_Assignor_Interface IS Job_Assignor_Module();
   SC  : array [integer] of Scheduler_Interface
       IS (1.. $Num_Controlled_Resources, .
   EX : array [integer] of Executor_Interface
       IS (1.. $Num_Controlled_Resources, .
   PS  : Plan_Selector_Interface IS Plan_Selector_Module ();

CONNECT

-- ** Connections Between Job Assignor and       **
-- ** higher-level Behavior_Generation Interface **


(?J : Task_Command_Frame)
Do_task (?J) ||>
  JA.Do_task(?J);

(?J : Task_Command_Frame)
JA.Fetch_task_frame (?J) ||>
  FWD_Fetch_task_frame (?J);

(?J : Task_Command_Frame; ?TF : Task_Frame)
RCV_task_frame (?J, ?TF) ||>
  JA.RCV_task_frame (?J, ?TF);

(?ST :string)
JA.JA_Status (?ST) ||> BG_Status (?ST);

-- ** Generated Connections Between Job Assignor and        **
-- ** Scheduler Interfaces & Between Scheduler and Executor **

For i : integer in 1..$Num_Controlled_Resources GENERATE

  (?J : Task_Command_Frame)
  JA.Schedule_Job(?J) ||>
        SC[i].RCV_Schedule_Job(?J);

  (?CR : Controlled_Resources; ?J : Task_Command_Frame; ?S :
Schedule)
  SC[i].REQ_Simulate_Schedule (?CR, ?J, ?S) ||>
```

```
       FWD_Simulate_Schedule (?CR, ?J, ?S);

  (?CR : Controlled_Resources; ?ST : String)
  SC[i].SC_Status (?CR, ?ST) ||>
        JA.SC_Status (?CR, ?ST);

 (?CR : Controlled_Resources; ?J : Task_Command_Frame;
  ?S : Schedule; ?ST :string)
  EX[i].SND_EX_Status (?CR, ?J, ?S, ?ST) ||>
        SC[i].RCV_EX_Status (?J, ?S, ?ST);

END GENERATE;

-- ** Connections Between Scheduler and          **
-- ** higher-level Behavior_Generation Interface **

 (?CR : Controlled_Resources; ?J : Task_Command_Frame; ?S : Schedule)
 RCV_Simulation_Failure_Notification (?CR, ?J, ?S) ||>
 SC [Get_Index (?CR)].RCV_Simulation_Failure_Notification (?J, ?S);

 (?CR : Controlled_Resources; ?J : Task_Command_Frame;
  ?S : Schedule; ?RS : Result)
 RCV_Schedule_Evaluation (?CR, ?J, ?S, ?RS) ||>
    PS.RCV_Schedule_Evaluation (?CR, ?J, ?S, ?RS);

 (?CR : Controlled_Resources; ?J : Task_Command_Frame; ?S : Schedule)
 PS.Post_Schedule (?CR, ?J, ?S) ||>
     FWD_Post_Schedule (?CR, ?J, ?S);

 (?CR : Controlled_Resources; ?J : Task_Command_Frame;
  ?S : Schedule; ?ST : String)
   RCV_VJ_Status (?CR, ?J, ?S, ?ST) ||>
      SC[Get_Index (?CR)].RCV_VJ_Status (?J, ?S, ?ST);

-- ** Connections Between Executor and            **
-- ** higher-level Behavior Generation Interfaces **

 (?CR : Controlled_Resources; ?J : Task_Command_Frame)
  EX[Get_Index (?CR)].Do_Sub_Task (?CR, ?J) ||>
      FWD_Do_Sub_Task (?CR, ?J);

-- ** Connections Between Plan Selector and  **
-- ** Scheduler Interfaces                   **

 (?CR : Controlled_Resources; ?J : Task_Command_Frame;
  ?S : Schedule; ?ST :string)
```

```
  PS.SND_PS_Status (?CR, ?J, ?S, ?ST) ||>
     SC[Get_Index (?CR)].RCV_PS_Status (?J, ?S, ?ST);

-- ** Connections Between Plan Selector and  **
-- ** Executor Interfaces                    **

 (?CR : Controlled_Resources; ?J : Task_Command_Frame; ?S : Schedule)
  PS.SND_Execute_Schedule (?CR, ?J, ?S) ||>
     EX[Get_Index (?CR)].RCV_Execute_Schedule (?J, ?S);


-- CONSTRAINT
-- NEVER
--    (?J : Task_Command_Frame) Do_task (?J) || JA.Do_task(?J);

END; -- ARCHITECTURE BG_MODUL
```

A.7.1.3 Behavior Generation Submodules

```
--
--  JOB ASSIGNOR, PLAN SELECTOR, ARRAY OF N SCHEDULERS &
--  EXECUTORS FOR CONTROL NODES AT NEXT LOWEST LEVEL THAT
--  ARE CONTROLLED BY THIS NODE
--
```

A.7.1.3.1 Job Assignor

```
--
-- ************************************
-- ** JOB ASSIGNOR INTERFACE      **
-- ************************************
--
TYPE Job_Assignor_Interface IS INTERFACE;
ACTION
IN
    Do_task (J : Task_Command_Frame),
    RCV_task_frame (J : Task_Command_Frame; TF : Task_Frame),
    SC_Status (CR : Controlled_Resources; ST : String);
 OUT
```

```
    Schedule_Job (J : Task_Command_Frame),
    Fetch_task_frame (J : Task_Command_Frame),
    Decompose_task_frame (TF : Task_Frame),
    JA_Status (ST : String);
BEHAVIOR
  Decompose_function   : FUNCTION (TF : Task_Frame);

  --
  --NOTE: Should this function return a list of Scheduler/job pairs?
  --      The exact definition still needs some attention.
  --
BEGIN

(?J : Task_Command_Frame)
Do_Task (?J) ||>
   Fetch_task_frame (?J);;

(?J : Task_Command_Frame; ?TF : Task_Frame)
RCV_task_frame (?J, ?TF) ||>
   Decompose_Task_Frame (?TF);;

( ?TF : Task_Frame)
Decompose_Task_Frame (?TF) ||>
   Decompose_function (?TF);;

CONSTRAINT
 --
 -- (1) Do not allow causally independent Do task
 --     and Schedule Job events!
 --
 NEVER
   (?J1, ?J2 : Task_Command_Frame)
   Do_Task (?J1) || Schedule_Job (?J2);
 --
 --
 -- (2) Do not allow causally independent Do task
 --     and Status Message events!
 --
 NEVER
   (?J1 : Task_Command_Frame; ?ST : string)
   Do_Task (?J1) || JA_Status (?ST);
 --
 --
 -- (3) Do not allow Do Task and Fetch Task Frame events
 --     to be causally independent.
 --
```

```
 NEVER
   (?J1, ?J2 : Task_Command_Frame)
   Do_Task (?J1) || Fetch_task_frame (?J2);
 --
 --
 -- (4) Do not allow a causally dependent pair of Do Task
 --     and Fetch Task Frame events for different jobs.
 --
 NEVER
   (?J1, ?J2 : Task_Command_Frame)
   Do_Task (?J1) -> Fetch_task_frame (?J2)
   WHERE ?J1 /= ?J2;


END; -- Job_Assignor_Interface
```

A.7.1.3.2 Scheduler

```
--
-- **********************************
-- ** SCHEDULER INTERFACE          **
-- **********************************
--

TYPE Scheduler_Interface IS INTERFACE;
ACTION
 IN
    RCV_Schedule_Job (J : Task_Command_Frame),
    RCV_PS_Status (J : Task_Command_Frame; S : Schedule; ST : string),
    RCV_EX_Status (J : Task_Command_Frame; S : Schedule; ST : string),
    RCV_VJ_Status (J : Task_Command_Frame; S : Schedule; ST : String),
    RCV_Simulation_Failure_Notification (J : Task_Command_Frame;
              S : Schedule),
    RCV_Check_Schedule_Consistent (CR : Controlled_Resources;
                                   J : Task_Command_Frame;
                                   S : Schedule),
    RCV_Schedule_Consistency_Evaluation (CR : Controlled_Resources;
    J : Task_Command_Frame;
    S : Schedule);
OUT
    REQ_Simulate_Schedule (CR : Controlled_Resources;
              J : Task_Command_Frame; S : Schedule),
    Checkif_Schedule_Consistent (CR : Controlled_Resources;
                                 J : Task_Command_Frame; S : Schedule),
```

```
    SND_Schedule_Consistency_Evaluation (CR : Controlled_Resources;
                                          J : Task_Command_Frame;
          S : Schedule),
    SC_Status (CR : Controlled_Resources; ST : String);
BEHAVIOR
   Resource_name : Controlled_Resources; -- the lower-level controlled
resource
END;
```

A.7.1.3.3 Executor

```
--
-- ************************************
-- ** EXECUTOR INTERFACE             **
-- ************************************
--

TYPE Executor_Interface IS INTERFACE;
ACTION
IN
     RCV_Update_Schedule (J : Task_Command_Frame; S : Schedule),
     RCV_Execute_Schedule (J : Task_Command_Frame; S : Schedule);
  OUT
     Do_Sub_task (CR : Controlled_Resources; J : Task_Command_Frame),
     SND_EX_Status (R: Controlled_Resources; J : Task_Command_Frame;
                            S : Schedule; ST : string);

BEHAVIOR
  Resource_name : Controlled_Resources; -- the lower-level controlled
resource
END;
```

A.7.1.3.4 Plan Selector

```
--
-- *********************************
-- ** PLAN SELECTOR INTERFACE     **
-- *********************************
--

TYPE Plan_Selector_Interface IS INTERFACE;
ACTION
```

```
IN
     RCV_Schedule_Evaluation (CR : Controlled_Resources;
                              J : Task_Command_Frame;
                              S : Schedule;
                              RS : Result);
OUT
     SND_PS_Status (R: Controlled_Resources;
              J : Task_Command_Frame; S : Schedule; ST : string),
     Post_Schedule (R: Controlled_Resources;
              J : Task_Command_Frame; S : Schedule),
     SND_Update_Schedule (R: Controlled_Resources;
              J : Task_Command_Frame; S : Schedule),
     SND_Execute_Schedule (R: Controlled_Resources;
              J : Task_Command_Frame; S : Schedule);

END;
```

A.7.2 World Modeling

```
--
-- ****************************************************
-- ** WORLD MODELING INTERFACE and ARCHITECTURE        **
-- ** INCLUDING SIMULATOR AND KNOWLEDGE_BASE COMPONENTS  **
-- ****************************************************
--
```

A.7.2.1 Interface for World Modeling

```
TYPE World_Modeling_Interface IS INTERFACE;
ACTION
IN
     RCV_Fetch_task_frame (J : Task_Command_Frame),
     RCV_Request_KB_Object (Obj : string),
     RCV_KB_Object (Obj : string),
     RCV_Simulate_Schedule (CR : Controlled_Resources;
                            J : Task_Command_Frame; S : Schedule),
     RCV_Post_Schedule (CR : Controlled_Resources;
                            J : Task_Command_Frame; S : Schedule),
     RCV_Update (Obj : string; Data : string);
  OUT
     FWD_task_frame (J : Task_Command_Frame; TF : Task_Frame),
     FWD_REQ_Evaluate_Schedule  (CR : Controlled_Resources;
                            J : Task_Command_Frame;
            S : Schedule),
```

```
     FWD_Simulation_Failure_Notification
         (CR : Controlled_Resources; J : Task_Command_Frame;
           S : Schedule),
     FWD_REQ_KB_Object (Obj : string),
     FWD_KB_Object (Obj : string),
     FWD_Predicted_Input (Obj : string);

CONSTRAINT
 --
 --     Do not allow RCV_Simulate_Schedule and
 --     FWD_Simulation_Failure_Notification events to be causally independent.
 --
 NEVER
    (?CR : Controlled_Resources; ?J : Task_Command_Frame; ?S : Schedule)
    RCV_Simulate_Schedule(?CR, ?J, ?S) ||
    FWD_Simulation_Failure_Notification (?J, ?S);
 END
```

A.7.2.2 *Rapide* Architecture for World Modeling

```
ARCHITECTURE World_Modeling_Architecture ()
  RETURN World_Modeling_Interface IS

SI : Simulator_Interface IS Simulator_Module();
KB : Knowledge_Base_Interface IS Knowledge_Base_Module();

CONNECT

-- ** Connections Between Knowlege_Base and  **
-- ** higher-level World Modeling Interfaces **

(?J : Task_Command_Frame)
RCV_Fetch_task_frame (?J) ||>
   KB.RCV_Fetch_task_frame (?J);

(?J : Task_Command_Frame; ?TF : Task_Frame)
KB.SND_task_frame (?J, ?TF) ||>
   FWD_task_frame (?J, ?TF);

-- ** Connections Between Simulator and  **
-- ** higher-level World Modeling Interfaces **

(?CR : Controlled_Resources; ?J : Task_Command_Frame; ?S : Schedule)
```

```
RCV_Simulate_Schedule (?CR, ?J, ?S) ||>
   SI.RCV_Simulate_Schedule (?CR, ?J, ?S);

(?CR : Controlled_Resources; ?J : Task_Command_Frame; ?S : Schedule)
SI.REQ_Evaluate_Schedule (?CR, ?J, ?S) ||>
   FWD_REQ_Evaluate_Schedule (?CR, ?J, ?S);

(?CR : Controlled_Resources; ?J : Task_Command_Frame; ?S : Schedule)
 SI.SND_Simulation_Failure_Notification (?CR, ?J, ?S) ||>
 FWD_Simulation_Failure_Notification (?CR, ?J, ?S);

END; -- Architecture
```

A.7.2.3 World Modeling Submodules

A.7.2.3.1 Simulator

```
--
-- ********************************
-- ** SIMULATOR INTERFACE        **
-- ********************************
--
TYPE Simulator_Interface IS INTERFACE;
ACTION
 OUT
    REQ_Evaluate_Schedule (CR : Controlled_Resources;
                    J : Task_Command_Frame; S : Schedule),
    SND_Simulation_Failure_Notification (CR : Controlled_Resources;
                    J : Task_Command_Frame;
                    S : Schedule),
    SND_Predicted_Input (Obj : string);
 IN
    RCV_Simulate_Schedule (CR : Controlled_Resources;
                    J : Task_Command_Frame; S : Schedule);
END;
```

A.7.2.3.2 Knowledge Base

```
--
-- ********************************
-- ** KNOWLEDGE BASE INTERFACE   **
-- ********************************
--
```

```
TYPE Knowledge_Base_Interface IS INTERFACE;
ACTION
  OUT
     SND_task_frame (J : Task_Command_Frame; TF : Task_Frame),
     REQ_KB_Object (Obj : string),
     SND_KB_Object (Obj : string);
  IN
     RCV_Fetch_task_frame (J : Task_Command_Frame),
     RCV_Request_KB_Object (Obj : string),
     RCV_KB_Object (Obj : string),
     RCV_Post_Schedule (J : Task_Command_Frame; S : Schedule),
     RCV_Update (Obj : string; Data : string);
END
```

A.7.3 Value Judgement

```
--
-- ********************************
-- ** VALUE JUDGEMENT INTERFACE  **
-- ********************************
--

TYPE Value_Judgement_Interface IS INTERFACE;
ACTION
 OUT
    SND_Schedule_Evaluation (CR : Controlled_Resources;
                              J : Task_Command_Frame;
                              S : Schedule;
                             RS : Result),
    SND_VJ_Status (CR : Controlled_Resources;
                                 J : Task_Command_Frame;
                                 S : Schedule;
                                 ST : String);
 IN
    RCV_Evaluate_Schedule (CR : Controlled_Resources;
                            J : Task_Command_Frame; S : Schedule),
    RCV_Predicted_Input (Obj : string),
    RCV_Update (Obj : string; Data : string);
CONSTRAINT
--
-- Do not allow causally independent receive evaluate requests
-- and evaluation outputs

 NEVER
```

```
   (?CR : Controlled_Resources; ?J : Task_Command_Frame;
    ?S : Schedule; ?RS : Result)
    RCV_Evaluate_Schedule (?CR, ?J, ?S) ||
    SND_Schedule_Evaluation  (?CR, ?J, ?S, ?RS);
END;
```

A.7.4 Sensory Processing

```
--
-- **********************************
-- ** SENSORY PROCESSING INTERFACE **
-- **********************************
--

TYPE Sensory_Processing_Interface IS INTERFACE;
ACTION
 OUT
    SP_SND_Update (Obj : string; Data : string);
 IN
    Observed_Input (Data : string),
    SP_RCV_SP_Data (Obj : string; Data : string),
    Predicted_Input (Data : string);
END;

--
```