SADL: Simulation Architecture Description Language

Kenneth G. Ricks
National Aeronautics and Space Administration
Marshall Space Flight Center

John M. Weir
National Aeronautics and Space Administration
Marshall Space Flight Center

B. Earl Wells
Department of Electrical and Computer Engineering
The University of Alabama in Huntsville

Abstract

This paper introduces the Simulation Architecture Description Language (SADL) developed at the National Aeronautics and Space Administration’s Marshall Space Flight Center to support the real-time simulation of advanced avionics systems. SADL is a graphical, domain-specific Architecture Description Language (ADL) that facilitates the high-level specification of both the software and hardware aspects of hard real-time avionics system simulations targeted for execution on diverse hardware architectures, including multiprocessor systems. It supports the hierarchical expression of the architecture of an entire simulation at various levels of abstraction. A detailed description of SADL is provided along with a case study that illustrates its ability to represent real-world simulations.

Key words: Architecture Description Language, real-time, simulation, software architecture, simulation model
1 Introduction

Real-time computation occurs when the computer’s internal concept of time is closely synchronized with actual time [1]. One area in which real-time computation is required is simulation of advanced avionic systems. This is a very demanding type of computation requiring complex calculations within small time frames. It requires state-of-the-art hardware to provide the computational speed required and special system software that is designed to minimize the overhead associated with program execution [2]. Real-time simulation plays an important role in the evaluation and testing of avionic systems. It provides a means of analyzing a design before it is built to uncover design flaws and test proof-of-concept early in the design process. Also, real-time simulation can facilitate the testing of existing hardware subsystems by imitating the real-time operation of subsystems interconnected to the hardware. In this way, the test article can be fully evaluated without the presence of the subsystem hardware [1]. Finally, real-time simulation can aid in project planning by providing virtual hardware. For example, before each space shuttle mission, simulations of the shuttle main engines are executed using the proposed flight profile to determine how they will perform. Any anomalies found in the performance are analyzed and the flight profile is modified to create an optimum profile [3]. In this case, it is impossible to use the actual hardware for such planning activities.

The Simulation Group of the National Aeronautics and Space Administration’s (NASA) Marshall Space Flight Center (MSFC) is responsible for the development of real-time simulations for avionic systems ranging from launch vehicles to on-orbit systems. This group has produced simulations in support of several NASA projects.
Earth-to-orbit simulations of the National Launch System (NLS) were developed to investigate design issues of the unmanned heavy-lift launch vehicle that was proposed to supplement the payload capabilities of the shuttle. Orbital systems simulated include the Orbital Maneuvering Vehicle (OMV), a proposed “space tug” used to remotely move orbiting hardware from one orbit to another, and the Space Station Furnace Facility (SSFF), a proposed space station payload to be used in crystal growth and other scientific experiments requiring microgravity environments. The Automated Rendezvous and Capture (AR&C) program is studying techniques to automate the docking of orbiting spacecraft. The recent collision between the Russian space station Mir and its re-supply vehicle demonstrates the importance of this effort [4][5]. Subsystem simulation is also performed to support hardware-in-the-loop testing of system components. Testing of the Global Positioning System (GPS) and Inertial Navigation System (INS) algorithms used for the Guidance Navigation and Control (GN&C) of the X-33 vehicle is an example. X-33 is a small-scale version of the Reusable Launch Vehicle (RLV) used to verify early concept design issues related to the full-scale RLV [6][7].

These simulation activities are supported by state-of-the-art real-time simulation facilities such as the Vehicle Simulation Lab, the Engine Simulation Lab, the Actuator Test Lab, the Flight Robotics Lab, and the Contact Dynamics Simulation Lab all located at the MSFC. These facilities are interconnected via high-speed optical links and support simulations of complete autonomous spacecraft missions including pre-launch operations, launch, orbital maneuvers, rendezvous, docking, and landing [8].

Although each project is different, the simulations of these avionic systems have similar requirements, characteristics and behaviors that can be used to define a simulation
model. A simulation model is a description of the simulation and represents specific simulation methods. It includes architectural information such as the types of hardware and software components in the simulation, the interfaces among components, and the software architecture. In addition to the architectural information, a simulation model includes implementation details such as the properties of the host computers and software execution characteristics. The simulation model developed at the MSFC is used from project to project to create real-time simulations of advanced avionic applications. However, in the past, ad hoc techniques were used to apply this model to new applications.

In order to formalize the application of the simulation model to target applications, a custom Architecture Description Language (ADL), called the Simulation Architecture Description Language (SADL), has been developed. SADL is a high-level design language used to specify the architecture of the entire simulation with various levels of abstraction. It is particularly designed to represent the simulation model developed at the MSFC.

2 Related Work

In the last several years the development of high-level software design tools targeted for real-time simulation software has gained increased attention. This research is an effort to move away from the ad hoc design paradigms of the past and to unlock the many benefits of high-level software design such as code reuse, abstraction of implementation details, documentation support, tool support for evaluation and analysis, formalized design processes, and a tight coupling between the design of the system and
the underlying model upon which the system design is based [9]. There has been much research done involving ADLs [9][10] and in the related area of hardware/software codesign [9][11].

The result of this research has been the development of several capable tools and design environments specifically targeted for high-level design of real-time simulation software. One such design environment is called DARTS/DShell and was developed by the Jet Propulsion Laboratory (JPL). It combines a flexible body dynamics computational engine based on a spatial algebra algorithm with a library of hardware models to provide a development environment for real-time, hardware-in-the-loop spacecraft simulation. It provides code generation capabilities and has a custom graphical user interface (GUI) for providing model descriptions as inputs [12].

The Prototyping Environment for Real-Time Software (PERTS) is another example of a real-time software design environment. PERTS is tailored to the analysis and evaluation of prototype real-time systems. The software description is provided as a task graph based upon the integration of the periodic-task model, the complex-job model, and the imprecise computation model. The computational resources are specified using a resource graph. Task scheduling, resource management, and performance analysis and profiling are provided within the environment [13].

Another product developed for real-time software design is MetaH. MetaH is an ADL that works in conjunction with ControlH, a domain-specific language and toolset for the design, analysis, and implementation of GN&C algorithms. MetaH takes a ControlH specification as input and performs real-time schedulability and reliability
analysis on it. The inputs to MetaH can be in graphical or textual form and must include a description of the software and the hardware used to execute the software [14].

There are many tools available other than those mentioned above. Many ADLs exist that support different computational models and software architectural styles. There are less formal tools that provide a means to specify architectural information of a system and also use various models and styles. The literature refers to such tools as description languages, specification languages, requirements languages, and case tools [9][10]. The examination of all such tools and their classification is beyond the scope of this paper. However, despite the number of available tools, it is difficult if not impossible to find an existing tool or environment that supports the computational model and architectural style of an existing simulation model. For example, DARTS/DShell does not support the high-level specification of the run-time characteristics of simulation software within the MSFC simulation model. PERTS requires a task graph and resource graph as inputs representing executable software and its required computing resources. However, these do not accurately represent the high-level architecture of the entire simulation since hardware devices that do not execute on the host computer are not supported. MetaH uses architectural descriptions particularly tailored to GN&C applications. This is helpful in the design of some of the simulation software, but it does not support a description of the overall simulation. Tools based upon a pure data-flow model do not support synchronization between processes not sharing data since data availability is the firing rule [15]. State machine models get very cumbersome when dealing with large systems with many possible states. This compatibility issue is exacerbated since the architecture of most complex software systems does not neatly match a single
architectural style. Instead, they involve some combination of styles forming a heterogeneous architecture [16].

3 The MSFC Simulation Model

The avionic systems simulated at the MSFC involve complex mathematical calculations to represent vehicle properties such as dynamics, GN&C, mass properties, orientation, and propulsion. These calculations require matrix operations, look-up table functions, and the solution of differential equations. Also, algebraic, transcendental, trigonometric, and multiple variable function operations may be required [1][3]. Many simulations also require support for hardware-in-the-loop. This requires interfaces to hardware devices such as actuators, sensors, three-axis motion simulators, and vehicle subsystems such as flight controllers and GPS receivers. This hardware must be driven at rates sufficient to maintain real-time operation and to provide the required accuracy to the mathematical equations representing the vehicle properties [1]. This requires a low-latency, high-speed communication mechanism in order to minimize the response time of the hardware devices.

The complexity of the mathematical calculations together with the strict timing requirements of real-time computation make the execution of uni-processor allocations of the simulation software difficult if not impossible. As a result, the simulation software is partitioned into a set of processes and allocated to multiprocessor computing platforms in order to take advantage of parallel processing techniques. For this model, a process is defined as an executable segment of code that is a schedulable entity on the targeted host computer requiring processor time, memory, and other computing resources. A process
includes the context information associated with it and can exist in one of three states, running, ready, or blocked [17]. All processes are static, existing throughout the life of the simulation [18].

In order to accurately represent the behavior of these applications, the processes must assume different behavioral characteristics. The mathematical calculations typically execute in tight loops requiring iterative methods. These loops must be repeated periodically in order to maintain the stability of the output [1]. The processes mimicking this periodic behavior do so by being invoked by a timer on the host computer system or through synchronization with other periodic processes. These processes have hard deadlines that must be less than or equal to the period. All periodic processes usually share a timing environment based on major and minor frame times derived from the execution times required for the processes and their frequencies. For this reason, all periodic processes must have frequencies that are harmonics of one another. This hard real-time software requires correct results within the allowable time frames in order to be considered correct. An error in either the functional operation or the timing requirements is typically catastrophic [13][18]. The behavior of this type of software is sometimes described as discrete-time as opposed to periodic [19]. In this paper, the term periodic is used predominantly.

In addition to the periodic behavior, simulation software must be able to handle the aperiodic events that arise during operation [13]. Some aperiodic events correspond to real world events within the simulated system. Simulated system faults such as an inoperable thruster fall into this category. Other aperiodic events are side effects of the simulation itself. Receiving input from the operator of the simulation, providing
information about the status of the simulation to the operator, and handling errors within
the simulation software are examples. At the MSFC, a software executive called
simDriver was created to handle all such housekeeping activities [20]. Processes that
handle aperiodic events are not usually invoked at regular time intervals. Instead,
execution is dependent upon aperiodic external events or conditions and there is no
specific minimum time interval between invocations. This type of software is often used
to model discrete-event classifications of system models [19]. These types of events are
necessary in the overall execution of the simulation, but the aperiodic processes handling
them typically have soft real-time requirements. If the operator does not receive a status
message within a specified time frame, the simulation does not have to be aborted.
However, more severe problems occur when software having soft real-time requirements
prevents the hard real-time software from meeting deadlines [18].

Both periodic and aperiodic processes terminate execution by sleeping. That is,
they block waiting for an event. This releases the processor that has been executing the
process. When the dependencies are satisfied, the process returns to the ready queue.
This blocking behavior is what differentiates both periodic and aperiodic processes from
the last type of process present in the simulation model, continuous processes.
Continuous processes can be used to service both periodic and aperiodic events. They
are invoked once at the beginning of the simulation and never suspend their execution by
blocking. Instead, they poll external events or conditions consuming all of the
computational resources of a processor. The benefit of this type of process is that no
overhead is incurred by context switches and synchronization latencies. It is essential for
simulations having small frame times where additional overhead can not be afforded.
The partitioning of the simulation into individual processes is usually done along functional lines. For example, all the thruster software may be collected into one process, the dynamics software becomes another process, and the data archiving software becomes yet another process. This produces a simulation with coarse computational granularity and limits the amount of parallelism possible. If timing requirements cannot be met with such a high-level decomposition, the processes are decomposed further producing a finer granularity, a larger number of processes in the process set, and a higher degree of parallelism. The process set is typically targeted for execution on a symmetric multiprocessor system. The tightly coupled multiprocessor platform is preferred over a distributed system due to its shared memory architecture and the virtual uni-processor interface presented to the user. Scheduling of the process set is done using a preemptive priority based system with static priorities. There are multiple off-the-shelf (OTS) real-time kernels available that include this type of scheduler. Scheduling often includes some fine-tuning done by hand in order to guarantee that all deadlines are met. This often includes adjusting priorities or the creation of processor farms dedicated to the execution of compute intensive processes.

The simulation software architecture is organized as a heterogeneous mixture of several common architectural styles. There are client-server relationships such as those between the control software and the simulation software. There are pure data-flow relationships implemented using shared memory data repositories as well as point-to-point data communications. Finally, there are control-flow relationships that govern the execution path through the simulation software and define execution constraints [16].
Figure 1 shows the high-level architecture of a simulation developed at the MSFC for a typical application. The example application is a simulation developed for the AR&C program. AR&C is a program designed to test the viability of automated docking of on-orbit vehicles using video guidance sensors and laser operated docking mechanisms [4][5]. Simulation software processes are shown as ovals, and the color indicates to which major subsystem of the simulation each process belongs. This figure should give the reader an idea of the complexity involved with these simulations and of the difficulties involved with representing the architectures of such simulations with existing tools.
Figure 1. AR&C Simulation architecture.
4 Description of SADL

SADL is an ADL specifically designed to represent the architecture of real-time simulations using the simulation model developed at the MSFC. An ADL is defined as a formal language that can be used to represent the high-level architecture of a software intensive system [9]. Fundamentally, an ADL must support design reuse and provide evaluation and analysis of designs early in the project’s life-cycle [10]. The formal specification has clearly defined semantics describing the architectural components and their interconnections. Toolsets extract information from the architectural specification and provide capabilities like evaluation, analysis, testing, and code generation. The ability to provide such functions is an integral part of an ADL, but the toolset that actually provides the functions is not considered part of the ADL [9][10].

SADL is a hierarchical language supporting multiple levels of abstraction. General high-level descriptions similar to the box-and-line diagram in Figure 1 are supported at the highest levels of abstraction. The specification of implementation specific details is supported through lower-level tools. This hierarchical design simplifies the porting of a simulation between computing platforms. The high-level specification requires no changes, but each platform requires different low-level tools to support the implementation details.

SADL is a graphical language. It uses a directed graph of nodes and arcs to represent the simulation architecture. The nodes represent the simulation components, and the arcs represent their interactions. A design is entered by using a custom drawing tool created using the Domain Modeling Environment (DoME). DoME was developed at the Honeywell Technology Center to aid in the prototyping and development of graphical
tools and specifications [21]. It supports the execution of Alter methods that can be used to enforce design rules and create artifacts such as code and documentation [22]. SADL also provides a means to describe non-functional properties associated with the simulation components such as execution times and frequencies that can be used for scheduling purposes. The result is a directed graph with semantics associated with each node and arc giving these graphical elements the required information needed to enforce the desired simulation model and provide for advanced functions such as code generation, design analysis, and process allocation and scheduling via a toolset. It is the semantics associated with the graph elements that differentiates a SADL specification from “dumb” box-and-line drawings such as that shown in Figure 1.

4.1 Node Semantics

The nodes of a SADL specification represent the simulation components that are either hardware devices or simulation software. Graphically, node types are differentiated by shape. External hardware devices are represented with a triangle. As previously mentioned, there are many types of hardware components that can be used in a simulation. However, their internal behavior is abstracted, and they are all considered as black boxes. As a result, all hardware devices are represented using the same node type.

Simulation software is represented by three different types of nodes that differentiate the types of processes supported by this simulation model. Each periodic, or discrete-time, process node is represented using a circle. Each aperiodic process node is represented using two concentric circles. Each continuous process node is represented as three concentric circles. SADL provides for the specification of non-functional
properties of all three types of processes. Such properties include execution time, frequency, processor assignment, and other information used to support allocation and scheduling.

There are two additional node types defined. They are simulation containers and boundary nodes. The simulation container is drawn as a rectangle and can represent an arbitrary collection of graph elements. The boundary node is drawn as a diamond and represents the interface to an arbitrary set of arcs. Both of these node types are used for abstraction purposes allowing the designer to implement hierarchical designs with varying levels of abstraction. Figure 2 shows all the node representations used in SADL.

![Figure 2. SADL node representations.](image)

### 4.2 Arc Semantics

Real-time simulations involve timing relationships that can be very complex. Specific time dependencies among the simulation components may include a predefined order of execution among the simulation software processes. In addition to the timing relationships, data dependencies exist among the components as well. In order to capture the high-level architecture of such a simulation, all of these relationships must be represented accurately. SADL uses directed arcs to connect the nodes in a manner that represents the data flow, control flow, and timing relationships among the simulation
components. The arc types include a communication container, a data transfer arc, a synchronization (sync) arc, and a synchronization-with-data (sync-with-data) arc. The communication container is used for abstraction purposes and contains an arbitrary collection of arcs to abstract communications for high-level diagrams. The remaining arcs are used to represent the details associated with communicating nodes. The following sections describe the detailed semantics associated with data, sync, and sync-with-data communications between the various types of nodes.

### 4.2.1 Data Communications

Figure 3 shows two periodic processes, A and B, interconnected with a data arc. This type of relationship is very common in simulations where a global data repository is shared among the process set [16]. This relationship means that process A produces some data that process B consumes. There is no implied order of execution for the two processes involved, and there is no restriction on the invocation rates of either process. Process A produces the data, and process B consumes the data when it is needed. If process A has not updated the data when process B consumes it, then process B proceeds using old data. Similarly, if A updates the data multiple times between reads of the data by process B, then B uses only the latest data. It is assumed that the critical regions of each process reading from and writing to the shared memory are protected using semaphores or spinlocks in order to provide mutually exclusive reads and writes of the data. Although Figure 3 shows a data transfer between two periodic processes, the semantics of the data transfer are the same for any types of nodes interconnected with a data arc.
4.2.2 Synchronization Involving Periodic and Aperiodic Processes

The behavior of synchronizing processes is much more complex. Collaborative synchronization methods involving several processes such as barriers are not supported. All synchronizations are point-to-point communications between two processes that are connected with a sync or a sync-with-data arc. The general meaning associated with this relationship is that the source process executes first, provides a synchronization mechanism to the destination process, and then the destination process can execute. The synchronization mechanism is the trigger for the execution of the destination process, and the destination process blocks or polls while waiting for the synchronization. The sync and sync-with-data arcs have two properties that help define the relationship between the processes connected. The first property is the release time of the synchronization relative to the execution cycle of the source process. The execution cycle is the time a process executes for each of its invocations. Therefore, the release time is the time within the execution cycle of the source process that the synchronization is sent to the destination process. The synchronization mechanism can be sent at the beginning or the end of the execution cycle. Sending it at the beginning allows the destination process to begin approximately at the same time as the source process, thus parallelizing the execution of
the source and destination processes. Sending it at the end of the execution cycle means that the destination process must wait for the source process to complete execution before it can begin. In effect, this serializes the execution of the source and destination processes.

The second property required for a sync or sync-with-data arc is the frequency. This property represents the frequency at which the source process sends the synchronization mechanism to the destination process. There are two distinct value ranges for the frequency. A positive frequency value means that the synchronization occurs at the frequency specified by the positive value. A frequency value of zero means that the synchronization occurs aperiodically.

### 4.2.2.1 Synchronization Between Periodic Processes

The specific behavior of two periodic processes synchronizing depends upon the frequencies at which these processes are invoked and the frequency at which the synchronization mechanism is transmitted between the processes. The release time of the synchronization mechanism only controls the serialization or the parallelization of the execution of the processes and does not change the underlying semantics involved with the synchronization.

Consider two periodic processes A and B with frequencies \( F_a \) and \( F_b \) respectively as shown in Figure 4A. One possible relationship is for \( F_a \) to equal \( F_b \), and processes A and B synchronize during every invocation. This relationship represents two processes executing in lock-step fashion and is commonly seen in producer-consumer applications. Since process A and B execute at the same frequency and every invocation of process B must synchronize with every invocation of process A, the frequency of the
synchronization mechanism, $F_s$, is equal to $F_a$ and $F_b$. Hence, process A generates the synchronization mechanism during each of its execution cycles, and process B is invoked when it receives the synchronization mechanism. Figure 4B shows the resulting execution timeline. Figure 4B assumes that the synchronization release time is the end of A’s execution cycle thus serializing the execution of A and B.

Suppose $F_a$ is greater than $F_b$ and each time process B executes, it must synchronize with process A. Since all periodic processes must have frequencies that are harmonics of one another, $F_a$ must be an integer multiple of $F_b$. Since B synchronizes with A every time B executes, $F_s$ is equal to $F_b$. This relationship is common in situations where one process must collect data at a very high rate and filter it before sending it to a process executing at a slower rate. This requires process A to contain the logic necessary to provide the synchronization mechanism to process B at the correct frequency. This relationship is shown in Figure 4C where $F_a$ is twice $F_b$. Figure 4C assumes that the sync mechanism has a release time corresponding to the end of the execution cycle of process A.

Finally, consider the case in which $F_b$ is greater than $F_a$ and B must synchronize with A each time A is executed. In this case, $F_b$ must be an integer multiple of $F_a$. Process A executes at its frequency and sends a synchronization mechanism to process B during each execution cycle. That is, $F_s$ is equal to $F_a$. Process B executes more often than A and requires the use of a system clock and internal timing logic in order to control its own invocation rate. However, process B must also synchronize with process A every $N$ invocations. Such a relationship is often used to synchronize timers from two computer systems to accommodate for clock skew. This relationship is shown in Figure
4D where process B synchronizes with process A every two invocations of process B. In this figure, the release time of the synchronization mechanism is the end of the execution cycle of process A.

Several points need to be made regarding these relationships. First, there are many more possible relationships among Fa, Fb, and Fs than addressed above. However, the cases discussed above constitute those that are nearly always encountered within the defined application domain. Secondly, SADL enforces the semantics described above and will not allow improper connections to be made. For example, it is not legal for two periodic processes to be connected using a synchronization mechanism that is aperiodic in nature. Also, the semantics described in Figure 4 can be extended to include periodic processes interconnected with a sync-with-data arc. The only difference is that the processes synchronize and pass data instead of just synchronizing. Finally, a single graphical representation can represent several different execution relationships. The same nodes and arc in Figure 4A represent three different execution relationships. In order to determine the exact execution characteristics of the processes, the properties of the processes and the arcs must be examined.
Figure 4. The semantics of two periodic processes synchronizing.

4.2.2.2 Synchronization Between Periodic and Aperiodic Processes

Consider the case involving a periodic process synchronizing an aperiodic process. An example of such a relationship is a periodic process representing the dynamics of a vehicle sending data to an archiving process that can only execute upon the receipt of the synchronization mechanism accompanying the data. Figure 5A shows an aperiodic process, B, that requires synchronization from a periodic process, A. By
definition, execution of an aperiodic process is dependent upon some external event or condition that occurs aperiodically. In this case, that external event or condition is the synchronization from process A. Process A must contain all the logic necessary in order to send the synchronization to B at the appropriate times. Process B simply blocks waiting to receive the sync from A. The sync must have a frequency of zero indicating that it is aperiodic in nature unless it is acting in cooperation with other sync arcs into the aperiodic process. The topic of multiple syncs into a single process is discussed in further detail later in this paper. Figure 5B shows the execution timeline for this case. It assumes that the sync is released at the end of the execution cycle of process A. The sync can also be used to parallelize the execution of both processes by assigning it a release time at the beginning of the execution cycle of process A.

The case of an aperiodic process sending a synchronization mechanism to a periodic process is considered illegal by SADL and is not allowed. An aperiodic process is not capable of generating a periodic synchronization mechanism.
Synchronization Between Aperiodic Processes

Figure 6A shows an aperiodic process, A, synchronizing another aperiodic process, B. The frequency of the synchronization mechanism must be set to zero to indicate that it is aperiodic. During execution, A decides whether or not to send the synchronization to process B. Process B blocks on the synchronization mechanism from A entering the ready queue when it is received. There is no regular pattern established regarding the execution of A and B. A executes aperiodically based upon its external dependencies, and B executes upon receiving the synchronization from A. Figure 6B shows the execution timeline for this relationship assuming that the synchronization release time is set to the end of the execution cycle of process A.
4.2.3 Synchronizations Involving Continuous Processes and/or External Devices

Continuous processes behave differently than do periodic or aperiodic processes. They begin execution at the beginning of the simulation, and they busy wait while testing for events or conditions as opposed to blocking. As a result, they never relinquish the processor, and they execute for the entire duration of the simulation. The receipt of a synchronization does not change the state of the process from blocked to ready. Instead, it changes only the control flow through the process. Also, the release time of a synchronization from a continuous process must be at the beginning of its execution cycle by definition. Continuous processes are only invoked once and execute until the end of the simulation. All other processes in the process set must execute in parallel with them.

Figure 6. Synchronization between two aperiodic processes.
The semantics of synchronizations involving continuous processes are very similar to those involving the other types of processes since continuous processes model the same periodic and aperiodic behavior in the simulation as the other processes but with a different implementation. Consider a continuous process, A, sending a sync to a periodic process, B as shown in Figure 7A. Let $F_b$ represent the frequency of execution for process B. If B requires a synchronization from A during every execution cycle, process A must send the synchronization mechanism at the same frequency as $F_b$. That is, $F_s$ is equal to $F_b$. This is shown in Figure 7B. A second relationship occurs when process B has an invocation rate faster than the required synchronization rate with the continuous process. The continuous process sends the synchronization mechanism at its required rate, and the periodic process receives it every N invocations. This is shown in Figure 7C where $F_b$ is twice that of $F_s$. These relationships produce nearly the same behavior as those involving two periodic processes as seen in Figure 4B and 4C.
Figure 7. A continuous process synchronizing with a periodic process.

If the source process is a continuous process and the destination process is an aperiodic process, the synchronization mechanism is aperiodic, represented by a frequency equal to zero. The continuous process generates the sync mechanism at irregular intervals based upon some internal logic, and the aperiodic process blocks until receipt of the synchronization.

Continuous processes can also receive synchronizations from periodic and aperiodic processes. If the source process is aperiodic, then the synchronization must be aperiodic also. If the source process is periodic, there are two relationships of interest. The first is when the source process generates the synchronization mechanism at a
frequency equal to its invocation rate, $F_s$ is equal to $F_a$. In this case, the continuous
process requires a synchronization from each invocation of the periodic process. The
second case occurs when the source process generates the synchronization at a frequency
less than its invocation rate. Specifically, $F_a$ is an integer multiple of $F_s$. This represents
a situation in which the continuous process requires a synchronization from the periodic
process every $N$ invocations of the periodic process. In both cases, the continuous
process simply polls for the synchronization mechanism. It is the responsibility of the
continuous process to verify that all synchronizations are recognized and that none are
missed due to polling activities for different events or due to computational activity. The
synchronization from a periodic process to a continuous process is shown in Figure 8.

A) SADL representation of a periodic process synchronizing with a
continuous process.

B) $F_s = F_a$

C) $F_a = NF_s$

Figure 8. Synchronization from a periodic process to a continuous process.
Finally, what is the behavior of one continuous task sending a sync to another continuous task? In this case, the sync can be aperiodic or periodic. The behavior of the destination is simply to poll for the syncs.

Despite their obvious differences, an external device behaves the same as a continuous process when considering its interaction with other simulation components using a sync or a sync-with-data. Refer to the previous discussion for the details related to such communications.

Finally, one more issue involving synchronization needs to be discussed. This issue deals with how multiple synchronization arcs into a single process are handled. Some existing tools provide a means of combining execution constraints using OR operations and AND operations [13]. SADL implements an OR operation by allowing multiple synchronization arcs into an aperiodic process or a continuous process only if the same synchronization mechanism is used for all such arcs. For example, a process can receive a synchronization via a signal from multiple source processes as long as a signal is used for every synchronization. This allows the destination process to wait for one mechanism and begin execution on the first occurrence of the mechanism regardless of its source. If a signal and a message queue were used to synchronize the same process, the blocking or polling strategies of the destination process would be complicated considerably. SADL does not support more than one synchronization arc into a periodic process. The reason for this is due to complexities introduced such as clock skew that affect the execution of the destination process.
5 Case Study

The following paragraphs provide a description of the SADL representation of a simplification of the AR&C simulation. Figure 9 shows the top-level design of this simulation architecture. The left portion of the window provides access to the drawing tools. The nodes are in the left column and the arcs are in the right column. The right portion of the window is the drawing pane where the architecture is actually constructed. This simulation is broken into three sections, a vehicle simulation section, a hardware interface section, and a housekeeping software section. This decomposition is different from that shown in Figure 1. Each section is represented using a simulation container, and the interconnections between the sections are abstracted by the use of communication containers. The goal of this type of high-level diagram is to provide a design overview by hiding the details.

Figure 9. Top-level description of the architecture of the simulation.
Each of the simulation containers in Figure 9 contain a subdiagram showing all the details associated with the components that make up that section of the design. The contents of the vehicle simulation container are shown in Figure 10. The hardware interface section is shown in Figure 11, and the housekeeping software is described in Figure 12. In each section, the nodes and the arcs representing the actual simulation architecture are shown. As described earlier, the node types are distinguished by shape, and the arc types are distinguished by color. Abstractions are still used at this level to mitigate the clutter that appears in each editing pane. For example, multiple input or output arcs from a single node are abstracted into one communication container, drawn in black. The boundary nodes, the diamond-shaped nodes, represent the interface between a communication container and its contents. All the arcs that enter or leave one section of the design are represented at the left and right edges of the window as boundary points and appear in the connecting section’s window as boundary points as well.

Figure 10. The vehicle simulation portion of the design.
Figure 11. The hardware interface portion of the design.

Figure 12. The housekeeping software portion of the design.
Now consider the execution characteristics of the simulation processes specified in this example. There is one continuous process, the RF Interface that is located in the Hardware Interface subsystem. This process executes on its own processor and sends a sync every 50 ms to the Dynamics model in the Vehicle Simulation subsystem. The Dynamics model is in the group of periodic processes, which also includes the IMU model, the GPS Statistical model, and the Displayer. Each of these processes executes within a major and minor frame environment established on the host computer. The major frame is 200 ms and is comprised of four 50 ms minor frames. These times are derived from the frequencies of the periodic processes. The Dynamics model executes every 50 ms and must block waiting for the sync from the RF Interface process before each of its invocations. The frequency of this sync is 20 hz, and its release time must be at the beginning of the execution cycle since the RF Interface task is continuous. Upon receipt of this synchronization, the Dynamics process executes and provides a sync-with-data to the IMU model at a frequency of 20 hz with a release time at the end of the Dynamics process’s execution cycle. The IMU model also executes at 20 hz and blocks waiting for the synchronization mechanism from the Dynamics model. Therefore, every invocation of the IMU model is serialized with an invocation of the Dynamics model via this synchronization mechanism, and every invocation of the Dynamics model must synchronize with the RF Interface process. This relationship is shown in Figure 13.

The GPS Statistical model and the Displayer process are also periodic, but they have no synchronization dependencies. These processes, which each execute at 5 hz, must contain the logic necessary to control their own invocation rates by interfacing with the operating system. In this example, both processes must execute once every major
frame. For purposes of simplicity, they are shown executing in the first minor frame of each major frame.

There are three aperiodic processes in the simulation. They are the SimDrvr process, the Archiver process, and the Thruster model. The Thruster model has one input, a sync from the on-board computer (OBC). It blocks on this sync and is scheduled for execution sometime after its receipt. Sync-with-data arcs are used for communication from the Dynamics model, the IMU model, and the Thruster model to the SimDrvr and Archiver processes. These two processes only enter the ready queue upon the receipt of the sync mechanism from one of the models. These syncs are released at the end of the execution cycle of the models thus serializing the execution of the models and these aperiodic processes. In cases where multiple syncs are inputs to the same node, all of the source processes contributing a sync must use the same sync mechanism. In other words, the destination process will block on a single sync mechanism, and the source processes are merely sending different instances of the same sync mechanism. In this way, the syncs are logically ORed by the destination process.

The case study only represents a subset of the design shown in Figure 1. However, it demonstrates that the hierarchical capabilities of SADL greatly reduce the clutter and confusion of high-level descriptions of such complex systems while providing more detailed documentation of the project. Additionally, the non-functional properties associated with the simulation components document implementation and execution characteristics of the simulation software that can not be captured using simple block diagrams.
6 Future Research

SADL is a formal, high-level ADL used to aid the MSFC in the design and implementation of real-time simulations. It provides an interface to a host of external tools that can further enhance the design process. These tools support future research in areas such as allocation and scheduling. Particularly, this research will concentrate on effective techniques to merge schedules involving periodic tasks scheduled using a cyclic executive and aperiodic tasks scheduled using a preemptive, priority based scheduling algorithm. Additional research is planned to investigate the effect different interprocess communication mechanisms have on overall system schedulability.
7 Conclusions

The engineers in the Simulation Group at the MSFC have developed a simulation model that can be used to design real-time simulation software for advanced avionics systems. This model consists of periodic and aperiodic software processes with different execution characteristics inter-communicating using synchronization mechanisms, data transfers, or both. This model has been successfully applied to numerous avionics systems. Several tools were evaluated in an attempt to find an existing tool capable of representing this simulation model. However, differences in the underlying architectural styles and computational models supported proved too significant to overcome. As a result, a custom ADL called SADL was developed.

SADL is a formal graphical language based upon the simulation model developed at the MSFC and used to specify the simulation architecture at various levels of abstraction. It is designed to provide an architectural specification of the entire simulation, including both the software architecture and the hardware devices. It supports both high-level box-and-line specifications as well as low-level implementation specifics describing the execution characteristics of the software. It is sometimes necessary to represent these low-level execution characteristics in order to document the behavior of the system. SADL also supports the specification of non-functional properties associated with the software components and the communication mechanisms of the architecture. These properties are used for documentation and tool support purposes such as the tools that provide schedulability tests. SADL is flexible enough to allow designers to use other simulation models. For example, by using a subset of the process types supported by SADL, an engineer can design a system consisting of all
periodic processes and apply standard schedulability tests such as those based upon Rate Monotonic analysis. Similarly, by manipulating some of the non-functional properties of the simulation components, various host computer architectures can be targeted for execution of the simulation software. Finally, SADL can interface with existing design tools to create an even more powerful design environment. Tools such as MetaH and MATLAB can be used to design particular pieces of the simulation software such as the GN&C algorithms. These pieces can then be integrated into the process set and considered like all other simulation software processes within SADL.
References


