The Spring System Description Language*

Douglas Niehaus, John A. Stankovic, Krithi Ramamritham
Computer Science Department
University of Massachusetts
Amherst, Mass. 01003

February 16, 1993

Abstract

The introduction of hard timing constraints to the semantics of a computation presents a set of design challenges that are significantly different from those of conventional systems, requiring the ability to *guarantee* that timing constraints are met. Such systems must consider the *worst* case, rather than the average case, behavior of the system and application software. The Spring system uses a scheduler constructing explicit plans for executing application programs to ensure that both logical and temporal constraints are satisfied. Such scheduling requires a detailed representation of the worst case run-time behavior of the application. Constructing the behavioral description requires detailed information about the properties and requirements of the application, system, and target hardware.

The Spring system description language (SDL), plays vital roles in both its source and compiled forms. SDL source files enable developers to specify the properties of all parts of the system in great detail. When compiled, this information is available for use by all tools needing to use, modify, or add to it. The SDL thus provides vital support for specification, compilation, and execution of applications on the Spring system, as well as for specifying input information to simulations run under Spring's scheduling testbed.

This document presents the many sections of the SDL in detail, discusses its roles within the Software Generation System, the use of the information it provides at run-time, and describes the set of interface routines available for use by tools wishing to use or modify SDL information stored in compiled form.

*This work was supported by ONR under contracts NOOO14-85-K-0398 and NOOO14-92-J-1048 and by NSF under grant DCR-8500332.

Contents

1 Introduction

Hard real-time systems such as nuclear power plants, space stations, avionics, and many process control applications require very careful design, implementation and evaluation to ensure that explicit and individual timing constraints are met. The introduction of timing constraints to the semantics of a computation presents a set of design challenges that are significantly different from those of conventional systems. Current designs at all levels of conventional systems generally concentrate on improving average case performance.

Real-time application software implemented within systems having an essentially conventional character *implicitly assumes* that the average case orientation of the system design and implementation will be sufficient to support adequate performance under all scenarios within which the system is expected to function. Yet, such average case design is fraught with difficulties and subject to catastrophic failure if used to support systems with hard deadlines. For example, the copy-on-write approach found in the MACH operating system provides significant performance savings on the average, but might cause high execution time costs at exactly the wrong moment, causing important deadlines to be missed. Numerous similar examples exist in other areas of operating system and hardware design including: virtual memory, IPC, scheduling, I/O, caching, pipelining, memory refresh and bus contention in multiprocessors. Systems requiring the ability to *guarantee* that timing constraints are met must consider the *worst* case, rather than the average case, behavior of the system and application software. Further, the worst case behavior must be *predictable*, since it must be available for consideration *before* a program executes.

Spring is a real-time operating system providing support for predictable execution of real-time application software. The system is *reflective* in that it uses information about its current state and predictions of worst case application behavior when deciding how to schedule computations. The Spring scheduler constructs explicit plans for executing application programs thus guaranteeing they will meet their deadlines, subject to some basic assumptions about the accuracy of the behavioral predictions and the failure modes of the system[7]. The Spring scheduler assumes a *task* based run-time representation of a computation, where a task is defined as having a known worst case execution time (WCET) and set of resources requirements[6].

The Spring system description language (SDL) provides a simple but comprehensive way to collect, use, modify, and exchange descriptive information at the level of detail required to write, translate, and predictably execute real-time *programs*. However, it is important to note that the SDL is *not* intended as a high level requirements language, specification language, or design method. Some of the information normally specified at these higher levels is represented by the SDL, but most of it addresses lower level details, including: a vocabulary to describe the structure and properties of application and system software, a way to specify the task based run time representation of a program's behavior, a set of statements describing the location of all software within the target hardware's memory, and a way to describe the important properties of the target hardware.

The SDL also serves the important role of effectively *defining the set of information required to support a predictable real-time system.* As such, it also represents a reasonable *target* for higher level design, requirements, and specification languages, since any such higher level approach would have to generate the detailed descriptions required for translation and

execution as defined and supported by the SDL. It is also possible that the SDL itself could be expanded to support one or more higher level interfaces in the future. Further, graphic user interfaces could replace the source language for specifying many types of information, including: target hardware architecture, network topology, and the assignment of processes to processors.

Finally, it is important to note that the compiled form of the SDL provides a standard method for exchanging this descriptive information among all parts of the system that use, modify, or add to any part of it during compilation, at run-time, or during system simulation. The SDL thus provides vital support for system execution and simulation, as well as a way to integrate and coordinate the development and use of a wide range of tools.

The SDL plays a central role in the Spring programming and run-time environments. The programming environment includes the Spring-C programming language and the software generation system (SGS). The SGS includes the compiler, linker, and related tools. The Spring-C language supports a relatively conventional programming model, describing computations as sets of processes, where a process is a single thread of control within an independent address space. The SGS is responsible for translating from the process based programming model to the task based run time representation required by the Spring scheduler. The translation method involves *predicting* the worst case run-time behavior of the application program *at compile time.*

The translation depends on significant restrictions to the programming language and practices as well as a wide range of information about the application process and target system. The SDL is used to specify the information required as input to the translation method. When made, the behavioral predictions are also described using the SDL and are included in Spring executable files produced for each process during compilation. The SDL information in the executables is then used by the system during process activation to build and initialize the run time data structures used by the scheduler as it constructs execution plans. The Spring-C language and the translation method are described elsewhere[2, 3].

The compiled descriptive information helps coordinate the development and use of the various parts of the Spring project by providing a common form within which to produce and exchange information. When used to describe properties of a real-time application and target system for simulation using Spring's scheduling testbed, most of the information used as input by the simulation, include the task group representation of a computation, is specified using SDL source files. In contrast, when describing an application program that runs on the real system, the SDL descriptions of the processes are included in the Spring-C source files of the programs, but the task group representation of their run time behavior is *generated*, in compiled form, during translation. However, tools using the task based description neither know nor care whether it was generated by compiling an SDL source file, or derived during translation.

The SDL thus provides relatively mundane, but vital, support for stepwise system development by decoupling the production of descriptive information in compiled form from its use. In the initial stages of system development, the developer must specify everything using SDL source files. Then, as portions of the SGS and related tools are improved, more and more information is derived rather than specified. However, since the portions of the system *using* the information always work with the compiled form, they are completely insulated from such changes.

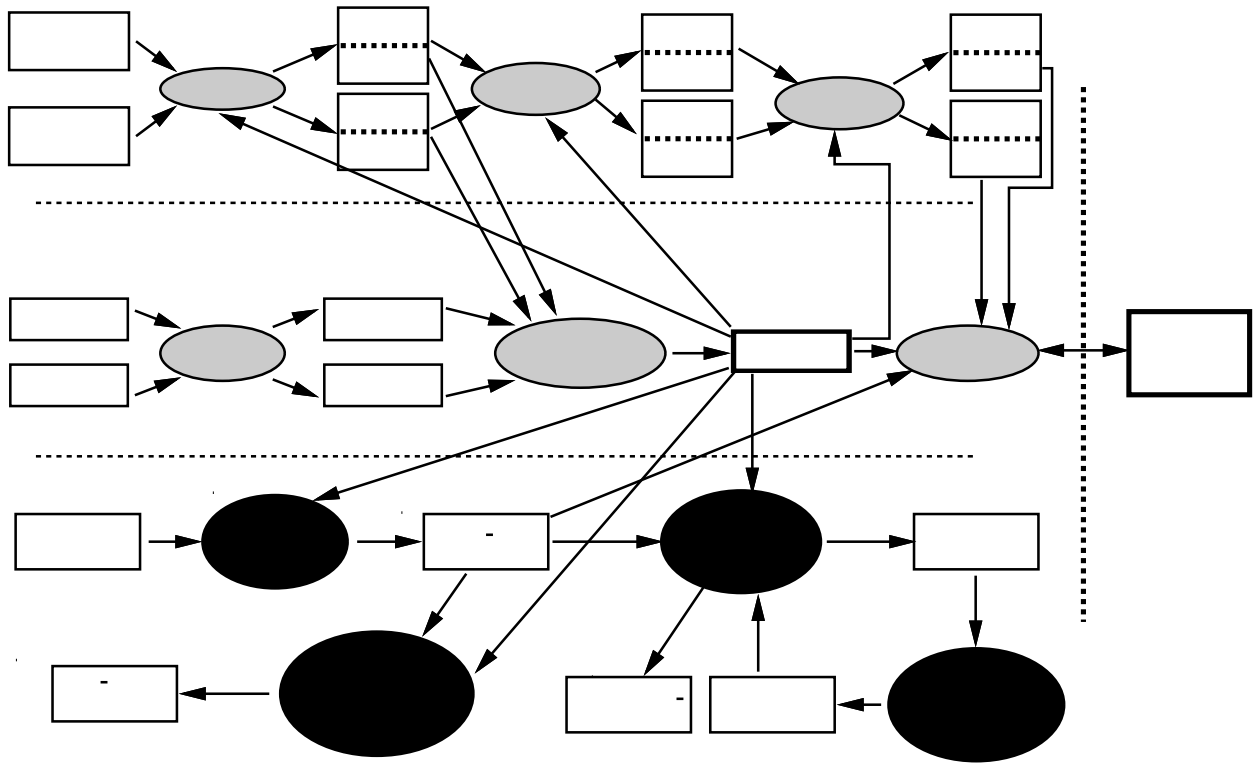


Figure 1: Programming Environment Information Flow

Section 2 of this document briefly describes the source and compiled forms of the SDL, and discusses in general terms how the SDL is used by various parts of the Spring system, particularly the programming and run-time environments. Section 3 discusses the source form of the SDL in detail, and Section 4 gives an example of how it can be used to describe real-time computations. Section 5 then describes the basic interface routines provided by the SDL library. This information is sufficient for anyone wishing to write a description for any portion of the system, or to write a tool using compiled descriptive information as input for analysis, or as a basis for deriving further descriptive information. Finally, Section 6 describes the current status of the SDL, and concludes the paper.

2 The SDL's Role in the Spring System

This section discusses the many roles of the SDL in the Spring system. Figure 1 illustrates the information flow in the Spring system, showing the central part played by the SDL. The figure is divided into three horizontal sections, illustrating the processing of Spring-C source files, SDL source files, and how the information compiled or derived is used to help control many aspects of system execution and simulation.

The middle section illustrates the processing of the source files which contain only SDL statements. This is emphasized by showing that the compiler used is **sdl_cc**, although **spr_cc** can also process files containing only SDL information. The SDL source files are compiled, and produce objects files containing only descriptive information. The **sdl_merge**

tool is then used to accumulate the descriptive information from the files as they are created, producing a full description of the system in the file *full.db*. This information is then available for use by several different parts of the system.

Much of the information derived from files written only in SDL is required to properly compile and link Spring-C programs, as illustrated in the top section of Figure 1. The boxes representing the object and executable files show that a Spring executable file contains a representation of the code, as well as the compiled form of the SDL information as specified by or derived from the Spring-C source during compilation. Examples of data supplied in SDL source files that have a bearing on the compilation and linking of executables include: the section specifying the target node structure, and the system layout section specifying the location of each process and shared segment within the target node. Further, information derived during compilation of some application source files is often required when compiling other source files.

For example, an application *process* will often be described using more than one source file. The main procedure of the process will often call procedures described in other source files. When compiling code for conventional systems this presents no problems, because each source file can be compiled independently of the others, and all unresolved references to routines or data structures in other files are resolved during linking. However, when compiling code for a real-time system requiring behavioral predictions during compilation, we must already have a behavioral description for all procedures called by the procedure being compiled. This implies constraints on how the procedures of a program are grouped into source files, and on the order in which the source files are compiled. The basic rule is that a given procedure can only be compiled when all of the procedures it calls have been compiled, and a description of their behavior is thus available.

This obviously implies that behavioral descriptions be included with all procedures within an object file archive, and that this information be available when a procedure is compiled, not only when the program executable is linked as is true in conventional systems. A reasonable approach would thus be to begin compilation of an application by adding behavioral descriptions of procedures in libraries used by the application to the *full.db* file. Then, as each source file is compiled, the behavioral descriptions for the procedures it contains are added to the *full.db* file.

Information about the procedures called is not the only information required to derive a behavioral description of a procedure during compilation. For example, just as vital for producing the behavioral description is the ability to answer the question: "how long will *this* memory reference take". How the information required to answer this question during compilation illustrates both the range of information represented by the SDL, and its central role in supporting program compilation and execution.

In a conventional uniprocessor system without caches memory reference time is usually a constant. However, many next generation real-time systems may be implemented using shared or distributed memory multiprocessors, some of which will have a non-uniform memory access (NUMA) hierarchy. A Spring node, for example, is a distributed memory multiprocessor with a two level memory hierarchy, as illustrated in Figure 2. Each node contains a set of single board processors which each have local memory. One of the processors is the system processor (SP), while the rest are application processors (AP). Each processor is connected to the node's VME backplane, giving each board access to the memory of the

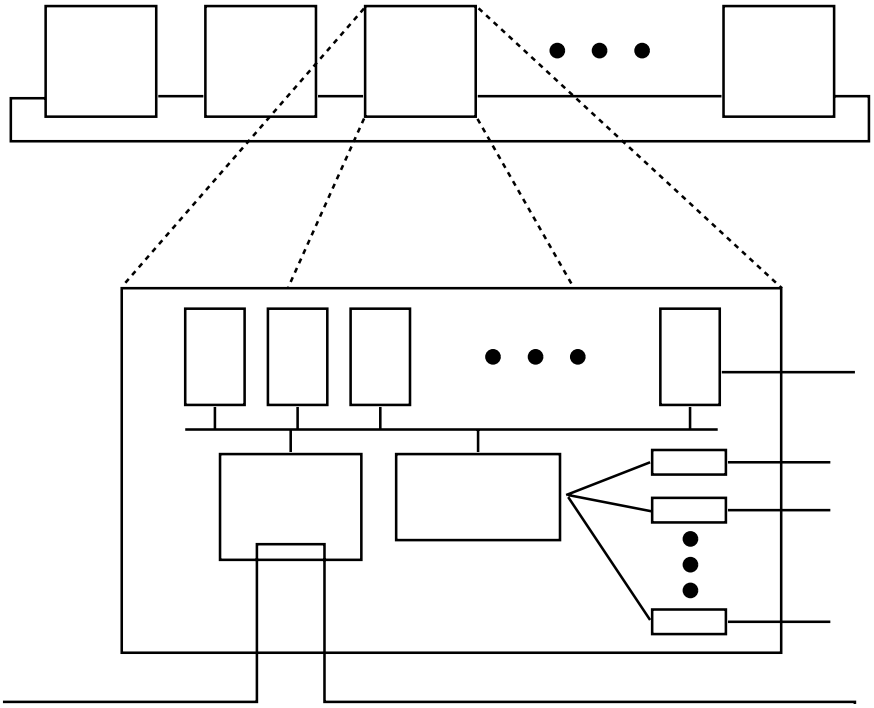


Figure 2: Spring Node Architecture

other boards, and to the global memory (GM) board connected to the bus.

The portion of the SDL described in Section 3.4 enables a developer to specify the structure of the node within which the application and system code will run. This description includes classification of each board as a memory or processor board, specification of its physical address range when address from the system bus, and categorization of its use. For example, a processor board can be used as an SP, AP or for I/O, while a memory board can be used to hold data, or represent a set of memory mapped control and status registers for an external device. Access by a processor to the local memory on the processor board, a *local* access, is faster than access by that processor board to the memory of other processors or to the GM board, a *global* access. This is not surprising since a global access requires use of the system bus with the associated arbitration overheads, while a local access does not. These are the two main levels of the NUMA hierarchy in the current Spring node architecture.

The *reflective memory* illustrated in the diagram, represents a potential third level to the hierarchy, although in the current configuration its access time is the same as to the GM board. The memory is "reflective" in that the reflective memory boards in each of the Spring nodes always contain the same information, subject to transmission delays. The reflective memory boards thus effectively represent a single memory board shared among all the nodes on the ring. An additional level in the NUMA hierarchy would be present in target hardware that contained instruction or data caches, which the current Spring node does not. Calculating at least a portion of the effects of instruction caches on the *worst case* behavior of a procedure is a non-trivial problem, but can be done under some circumstances [4].

The memory access time is ultimately determined by the location within the memory

hierarchy of the data structure being accessed relative to that of the process accessing it. Knowledge of the node architecture is only a part of the information required to determine memory access time, but the SDL represents all of the required information. Another important factor is the location of each process within the node, which is described by the layout part of the SDL described in Section 3.6.

Determining the the location within, the NUMA hierarchy, of each data structure at compile time is a bit more complicated, requiring information from several parts of the SDL. The executable image of a Spring process is compiled using the conventional model of a contiguous area within the address space of the process beginning with the process's text section, followed by the data section. The stack page is placed at the top of the address space. Each of these areas is private to the process, and each is supported by memory local to the processor board on which the process is activated, as specified in the layout specification.

If a process requires access to any *shared* data, or to the memory mapped control and status registers of an external device, the relevant shared memory segments are *attached* to the address space of the process during process initialization at boot time. In conventional systems the creation of shared segments and attaching them to a process's address space is done only at run-time, using system calls. While Spring provides system calls for this purpose, it also provides an interface with a compile-time component with several advantages for supporting shared segments containing pointers and predicting memory access time.

In the Spring system, shared memory segments can exist as independent "objects" at both compile and run times, and are specified using separately compiled source files. This makes it possible to use shared segments in a number of different ways, which are discussed in detail elsewhere [5]. For the purposes of this discussion, it is sufficient to note that when compiling a procedure, the compiler must be able to distinguish references to private data from those to shared data, it must know how shared data are grouped into segments, and where those segments are located.

Distinguishing references to private and shared data in a completely general way that accounts for the use of pointers is extremely complicated, and will require significant modifications to the Spring-C compiler beyond those completed or being implemented. A general solution will include extensions to Spring-C typing which will distinguish pointers to private and shared data, significant modifications to the compiler's intermediate code representation and to the assembler it emits to support annotations distinguishing private and shared access, and extensive modifications to the portion of the compiler making behavioral predictions.

However, we begin by supporting a less general, but simpler, method of access to shared data that supports distinguishing private and shared data accesses fairly easily. This simpler approach assumes that all shared data are accessed *by name*, rather than by pointer. Under the current method, shared data structures are grouped into sets represented as *resources*. These are the resources whose use by tasks in the group representing a process at run-time is of concern to the Spring scheduler. Resources can also be used to represent other types of objects or scheduling constraints, but are most often used to represent shared data.

A resource is defined using the portion of the SDL described in Section 3.2, which includes a list of the shared data structures represented by the resource, as well as its name, modes of access, and its association with a shared segment. Resources are grouped into sets, and assigned to a shared segment. The portion of the SDL described in Section 3.3 supports the definition of shared segments, which includes giving it a name, specifying a virtual and

physical base address, and listing the resources it supports.

Now consider the flow of information illustrated in Figure 1, with respect to determining memory access time, and knowing the behavioral descriptions of all called procedures, during the compilation of a given procedure. Several constraints on the order of compilation should now be evident. First, the node architecture and software layout specification, illustrated in the middle section of the Figure, must be compiled, and merged into *full.db*. Then, information about the behavior of library routines used by the application should be merged into *full.db*. This prepares the SGS to begin compiling the application source files, which should begin with the compilation of the files defining the shared segments, and merging their SDL descriptions into *full.db*. This provides the information required to classify as shared or private, all references to data that are made by name, since the layout specifies the shared segment location, the shared segment definition lists the resources it supports, and the resource definition lists the data structures it represents.

The compilation of the application procedures can then begin, starting with those that call only library procedures, since these are the only ones for which behavioral descriptions are known. Behavioral descriptions of these procedures are added to the set in *full.db* as they are produced, enabling subsequent compilation of procedures calling them. In this way, all of the application procedures can be compiled from the "bottom up". During compilation of a procedure, its access to global data structures is detectable since they are made by name. However, the access time is also a function of the location of the procedure, which is determined by the process within which it executes. The association of the procedure providing the processes' entry point is obvious, but it is less so for other procedures. The solution used at the moment is to assume a global access time for all shared data references, except those from the entry point procedure. In the case of the entry point procedure, the SDL provides information enabling the compiler to know if the process making the access and the shared segment being accessed reside on the same board. As we gain experience with application code, support for associating a particular procedure with a process or set of processes can be added to Spring-C and the SDL.

When all the application code is compiled, then executable files can be produced for each process. SDL information about shared segments accessed by a process is used by the loader **spr_ld** to resolve references to data structures supported by the segments. Executables for independent processes are complete when linked, and contain all of the SDL information required to build the run-time data structures describing them to the Spring scheduler. However, for processes that are part of a group engaging in synchronous communication, some post-processing is required to complete the SDL description. The **spr_grp** command reads the SDL information contained in the executable of each process in the group, and performs the analysis to write the complete descriptive information back out to the executable files. The analysis performed is described elsewhere[3].

If only application code is being produced, then *full.db* and the set of process executables are all that is required by the debugger **sbug**, which downloads the system and application code onto the Spring node(s) as described by the layout section of the SDL. The vertical dashed line symbolizes the fact that the download operation crosses the system boundary separating the development machine from the Spring target node. However, the SDL can also be used to provide input information to simulations associated with the Spring system. This information can describe actual application software, or describe an imaginary workload.

The bottom section of Figure 1 illustrates the workload generator, scheduling simulation, and Spring Scheduling CoProcessor (SSCoP) simulation testbeds. The workload generator uses the information in *full.db*, as well as a model of the workload, to generate a detailed system workload for specific experiments. If the workload is intended to drive the actual system, then **sbug** takes care of providing the workload to the running system as required. However, the system description and workload can also be used as input to the scheduling simulator to conduct a scheduling experiment. The scheduling simulation testbed enables researchers to experiment with different scheduling algorithms within a context that accurately reproduces many aspects of the actual Spring system [1].

The workload generated, and the information in *full.db* can also be used as input to the simulation of the SSCoP, which is a coprocessor designed to provide hardware support for Spring's explicit execution plan construction approach to system scheduling. The SSCoP interface code embodies algorithms and code that are drawn from the Spring system subsections addressing both process activation and the system scheduler. It uses the information in *full.db* to construct the run-time data structures used by the system, while the workload specifies the order and timing of the scheduling operations required. The SSCoP interface considers each set of computations requiring scheduling, and performs a number of preprocessing steps required to prepare the information in the form required by the SSCoP. This is then given to the simulation of the coprocessor, which gives the output back in a form requiring some postprocessing. The SSCoP interface code then uses the coprocessor output to update the system schedule as required, and records the results of each part of the simulation.

The two testbeds can be used on the same input information to compare the results produced by a simulation of a scheduling algorithm, and by the SSCoP. This is an important part of the testing process for the coprocessor. Further, results of the simulations can be compared to the behavior of the system actually using the SSCoP, once it is implemented.

This completes the discussion of how the SDL information is accumulated and then used by various portions of the Spring system. Most of the information flow illustrated in the figure, and discussed in this section, is enforced by proper use of the **make** command. When the developer is writing the makefile, he or she is responsible for making sure that the various processing steps are done in an order ensuring that information is available when required. However, while a bit more complex than those for conventional systems, the ordering constraints on the compilation steps are no different in principle from those requiring that all the object files required to build an executable be produced before linking occurs.

While they illustrate the central role played by the SDL, the details of exactly what information is used by which tools is less important than the idea that the descriptive information included in each file, and accumulated for the system as a whole, is stored in a common form, thus making it available to all present and future tools in the system requiring any part of it. The uses of the SDL information are by no means restricted to those discussed. For example, the download function of **sbug** was considered, but the descriptive information is used during debugging as well. Other tools will be developed over time which use the descriptive information, and may derive much of what is now explicitly specified by the developer.

One example of this would be a tool which would derive a layout for the system, taking

```
config_info  ::=   [conf_item]* system_layout [conf_item]* network_topology [conf_item]*
               |   [conf_item]* network_topology [conf_item]* system_layout [conf_item]*
conf_item    ::=   node_desc
               |   process_desc
               |   process_group_desc
               |   resource_desc
               |   shared_seg_desc
               |   task_group_desc
```

Figure 3: System Description Language Top Level

process requirements and properties into account. Some of the requirements information might be given by or derived from a higher level specification in a separate requirements language or requirements extension to the SDL. Such a tool would use the information about processes to discover a layout for the system which could make it easier for the system to fulfill the specified requirements, or it might seek to minimize IPC traffic by grouping processes that communicate onto the same node or even the same processor.

Other tools might conduct analyses related to specified fault tolerance requirements, and produce a process group specification appropriate to the level and type of fault tolerance required. The SDL is thus intended to support the orderly evolution of the system by serving as a target for higher level analyses, and by supporting a common format for the use and exchange of all relevant information. Further, the definition and implementation of the SDL makes it easy to add to or modify descriptive information. The interface supporting the use and modification of the SDL information by Spring tools is described in Section 5.

3 SDL Source Language

As a foundation for development, the SDL provides a significant portion of the framework within which a predictable real-time system can be established, and then gradually refined. Figure 3 shows the high level structure of a system description. The SDL grammar permits an arbitrary stream of language elements with a single instance each of the system layout and network topology specifications at any place in the stream. This is done to preserve flexibility in processing the application source files, since portions of the system description may appear at many locations within a source file, and can thus be processed by the SDL compiler in almost any order. However, it is reasonable to require a single system layout specification and a single network topology specification within the stream of definitions, as the grammar indicates. It also is important to note that the developer is not necessarily limited to providing input in source form. As development of the system proceeds, it is possible to imagine creating a graphic user interface for specifying some classes of information, including: target hardware architecture, network topology, and software layout. This interface would then produce the corresponding SDL description in compiled form, for use by other parts of the system.

The descriptions of individual items are compiled and combined by the SGS as it processes

each of the source files, forming a cumulative system description, as discussed in Section 2 and illustrated in Figure 1. The system description file holds the SDL in compiled form, and is thus readable by all the elements of the Spring programming and run-time environments (compiler, linker, system loader, and the debugger) requiring any of the information it contains. The information describing individual computations is included in the executable files describing their run-time representation to the system. These descriptions are read by the system as it activates processes during system boot.

Since it plays many roles, the SDL has several fairly distinct sections. One of the more interesting is the SDL's support for describing computations in terms of processes, process groups, and their run-time representations as groups of precedence related tasks. The SDL also supports the description of the resources and shared memory segments in the system, which are used by processes and considered by the system scheduler. A section of the SDL provides for describing the target nodes onto which the software must be loaded. This includes a description of each processor and memory board within each node, and the topology of the network connecting the nodes. The last part of the SDL specifies the system layout. This entails listing every process, resource, and shared memory segment that is assigned to each processor or memory board within each node.

3.1 Computation Descriptions

The specification of computations for a real-time system, or any system, can be done in a myriad of ways. In the Spring system we have adopted the *process*, a single thread of control through an independent logical address space, as the basic unit of computation. Each process is represented to the scheduler as a group of tasks, related by precedence constraints. A computation can be implemented as a single process, or as a group of processes. The structure of the process group is specified using precedence constraints, as with the task group. Process groups are useful for representing coarse grain concurrency within a computation, as well as describing structures used to support fault tolerance.

Processes communicate with one another either through messages or through shared data structures. Messages are sent and received through *ports*, using the interprocess communication (IPC) facilities. Asynchronous communication is not represented in the SDL, since it has no impact on the run-time representation of a computation, or on how it is scheduled. Shared data structures represent address space *overlap* among the processes doing the sharing, since the shared structure is visible in each of their address spaces.

When specifying the structure of a group using precedence constraints we use the simple, but effective, representation called a successor list. The list has a *Begin* node which is always the first item, but represents no part of the computation being described. The reason for this is that the group might contain several members that can execute concurrently from the very beginning. For each item in the group, we list its successors. If the item has no successors, then the imaginary *End* node signifying the end of the computation is implicitly assumed as its successor. Figure 4 specifies the grammar for a successor list.

The basic structure of the list is obvious, although the grammar is slightly complicated by the fact that successor lists can be constructed for groups of tasks, groups of processes, or groups containing process groups. The simplest form of a *list_item* identifies another element of the same group by name. However, it is also necessary to specify precedence

```
succ_list     ::=   succ_begin [succ_item]*
succ_begin    ::=   Begin: prec_list;
succ_item     ::=   name: prec_list;
prec_list     ::=   prec_item
                  | prec_list, prec_item
prec_item     ::=   list_item
                  | (list_item delay_val)
list_item     ::=   proc_group_name
                  | proc_name
                  | (proc_name task_name)
                  | task_name
delay_val     ::=   INTNUM
                  | (Comm_delay port_name_list)
                  | (Comm_delay INTNUM port_name_list)
```

Figure 4: Successor List

constraints that cross task group boundaries, since they are required to represent synchronous communication. A member of a group is uniquely identified by the pair giving the name of the group and the name of the group member. Precedence constraints can also have a delay value associated with them. The simple form for the delay value is as an integer number expressed in the basic time units of the system. This is useful for describing delays resulting from the use of the *delay* statement in Spring-C.

However, delays can also be created by synchronous communication, which is more complicated, since delay associated with a communication channel is only known at boot time. This is why the grammar permits expressing a delay value in the communication delay form to specify a list of port names. More than one port name is permitted since the delay between tasks in a group may represent more than one communication act. This is a consequence of the translation described in [3]. Further, a given precedence constraint may represent the delay associated with both communication and *delay* statements, and so permits an integer argument. When communication is involved, the actual delay value used is determined at process activation or scheduling-time by taking the maximum of the delays associated with all of of the ports involved, and that arising from *delay* statements, if any.

For example, consider a group of two processes engaging in synchronous communication using port A. The task group representing the processes will contain a precedence constraint between a task in the sending process and a task in the receiving process arising from the **sync_send** call, which sends a synchronous message on port A. Since the sending process should not continue execution until the message can make it to the receiving side, the delay associated with the precedence constraint would be: *(Comm_delay A)*. However, the communication delay can only be evaluated when the locations within the network of the sending and receiving processes using port A are known, so that the worst case delay associated with the connection can be evaluated. The SDL description of the task group in the process executables thus contains the *(Comm_delay A)* form, and is evaluated during process acti-

vation. The value for the delay is associated with the precedence constraint in the run-time representation of the task group used by the Spring scheduler.

At several places in the description of a computation, we wish to support the specification of *both* actual application code, *and* of a simulated workload based on that application. As a result, the SDL supports language elements of the *distribution* type. The distributions are used to specify certain values; particularly execution times, deadlines, and laxities. These are obviously important when specifying a workload for simulation, but have in some cases also been defined as specifying worst case values. The reason for this is that some distribution types, technically speaking, place no limit on the maximum value of the random variable. In such cases, the values in question play a slightly different role in the simulation where a distribution is required, and in the actual system, where a maximum value is needed. The distributions supported include the constant, uniform, exponential, and normal distributions. The grammars for the distribution types and those of other simple language elements such as names, numbers, and lists are specified in the complete SDL grammar in Appendix A.

3.1.1 Processes

The specification of a process is divided into three parts, those parameters related to its *execution*, those related to its *timing*, and those related to *scheduling* an instance of it. The execution parameters specify what file contains the process's executable image, what data structures it imports, what shared memory segments it uses, and the ports through which it engages in synchronous communication. The list of imported data structures specifies precisely what shared data structures are used by the process. These shared structures will clearly reside within one or more shared memory segments, but the process may use other segments which do not contain explicitly exported structures.

As discussed in Section 2, the classification of a data structure as being shared or private is an important part of determining the time required to access it in a NUMA architecture. While the system currently determines access time properly for shared structures that are referenced by name and explicitly exported, this will change. When it does, it will be reasonable for a process to attach a segment containing no exported symbols, since access time calculation for references through pointers will be supported. Thus, the list of imported data structures and that of shared segments are useful, since we clearly wish to know both sets of information. Although the shared segment list is currently redundant, since the sharing of the segments is implied by the import list, this is tolerable since it provides a way to help check consistency.

The specification of communication ports only lists those used for synchronous communication because synchronous communication has an impact on the run-time behavioral representation and scheduling of processes, while asynchronous communication does not. In the Spring system, any two processes engaging in synchronous communication are considered parts of a single computation, and will be represented as part of a task group representing the process group of which the communicating processes are members. The communication ports are represented at this level in the SDL because each process attaches to a port and then sends to or receives from it. At the process group level, one aspect of correctness is that the use of the synchronous communication ports be consistent. In this context, consistent means that every send on a port in a process will have a corresponding receive on the port

```
process_desc  ::=   Process(name) { [proc_attr]* } ;
proc_attr     ::=   exec_spec;
                  | timing_spec;
                  | sched_spec;

exec_spec     ::=   Code name
                  | Import name_list;
                  | Sharing name_list;
                  | Sync_ports port_list;

timing_spec   ::=   Non_Periodic;
                  | Period INTNUM;
                  | Periodic;
                  | Separation INTNUM;

sched_spec    ::=   Deadline dist_spec;
                  | Deadline_type dln_type;
                  | Importance INTNUM;
                  | Laxity dist_spec;
                  | RT_type rt_type;
port_list     ::=   port_item
                  | port_list, port_item
port_item     ::=   name
                  | (name use_type)
use_type      ::=   Receive
                  | Send
```

Figure 5: Process Specification

in the process with which it is communicating. Representing synchronous port use in the SDL enables the SGS to detect some classes of synchronous communication errors at compile time, including inconsistent use of the ports.

The timing specification is simple, identifying the process as periodic or nonperiodic. If it is periodic, then it gives the process's period, and a minimum separation constraint that must hold between successive instances. The minimum separation constraint is useful with nonperiodic processes as well, for simulations generating workloads. In that case it specifies the minimum separation between aperiodic events.

The scheduling parameters specify the process's deadline value, type of deadline, and laxity. They also give the process's importance and type. The sections of the grammar for these parameters are given in Appendix A. Deadlines may be hard, soft, or non-real-time, while processes are critical, essential, and non-essential. Note that not all combinations of process and deadline type are allowed; the combination of a critical process with a non-real-time deadline is, for example, nonsense.

```
task_grp_desc   ::=   Task_group(proc_name) { tg_def } ;
tg_def          ::=   group_def [task]+
group_def       ::=   Group_list { succ_list } ;

task            ::=   Task(name) { [task_attr]* } ;
task_attr       ::=   M_time dist_spec;
                    | Non_Preemptive;
                    | Preemptive;
                    | Resources [ru_spec]+;
                    | sched_spec;
                    | W_time dist_spec;

ru_spec         ::=   (name [use_attr]+)
use_attr        ::=   Start time_val;
                    | End time_val;
                    | Exclusive;
                    | Sim_prob use_prob excl_prob;
                    | Shared;
use_prob        ::=   FLOATNUM
excl_prob       ::=   FLOATNUM
```

Figure 6: Task Group Specification

3.1.2 Task Groups

A task group represents the worst case run-time behavior of a process which is used by the scheduler when constructing an execution plan. The grammar used to describe a task group is given in Figure 6. When a task group is declared, it specifies the name of the process it represents. The task group definition is separated from the process definition in the grammar as an aid to implementation. This is important because for application code, the description of the process will appear in its source code, while the task group representation will be generated by the compiler as it is derived during translation. However, we must *also* provide the ability to specify a task group in the SDL to support scheduling simulations. While we are interested in simulating application code, we are also interested in constructing simulations for workloads that may *not* represent actual application code.

The name specified when defining a task group is the name of the process it represents. The definition of a task group has two parts. The group definition gives the structure of the group using a successor list. Following that is a description for each of the tasks that appear in the group definition. Each task has a name, which need only be unique within the group. The scheduler, when working with all the tasks on the system can uniquely identify a task by the process name and task name pair. Note that for processes engaging in synchronous communication, the successor list of a given task group contains references to tasks in the group representing the other communicating process. Such references are of the

form ((proc_name task_name) (Comm_delay port_name)) mentioned in Figure 4. This form of the constraint says that the successor of the task listing it is the task name *task_name* within the group representing the process *proc_name*, and having a delay associated with communication through the port *port_name*.

The task has the same scheduling parameter specification *sched_spec* as a process or process group, which includes the deadline, real-time type, and so on. When a simulation is being run which uses computations represented by single tasks, these values can be used directly. When used to describe a process that runs on the Spring system, the use and meaning of the *sched_spec* items depends on the particular scheduling algorithm being used. For example, some scheduling algorithms require intermediate deadlines to be specified for each task in the group during a preprocessing phase executed after creation of the process's executable, but before scheduling the task group at run-time. Other scheduling algorithms may assume that the deadline of each task is that of the process or process group within whose representation it appears.

The use of the worst case execution time is obvious. A mean time is also present to aid in simulation studies. Both execution times are represented by distributions to aid simulation, although only a constant value for the worst case time is meaningful for software executing on the real system. As development of the system proceeds, we also envision expanding our representation of the WCET. One such extension would represent WCET as a function of one or more inputs to a computation known at scheduling time.

The specification of the resource use for a task has two forms. The form used to describe resource use of tasks representing real programs, and a form used only for simulations. The form representing programs gives the resource name and its mode of use, which is either *shared* or *exclusive*. This form can be used for simulations as well, of course, but the other form can *only* be used for simulations. It gives the name of the resource, the probability that the resource will be used and, if used, the probability that it will be used in exclusive mode.

These declarations all assume that the resource in question will be used for the duration of the task's execution, but that may not be true. The other two parameters make it possible to specify, relative to the beginning of the task, the earliest time at which the use of the resource can *start*, and the latest time at which it can *end*. This information has the potential to increase the schedulability of a task set, since it enables the representation to specify resource use closer to the actual process behavior, than if they did not exist. They are particularly useful for representing nested resource use blocks. However, resource use must be derived by careful analysis of the application code, as described in [3]. The resource use information is available to the scheduler in the run-time data structures describing the task group, which are built during process activation from the SDL information in the processes' executable files.

3.1.3 Process Groups

The processes as defined can be used to construct process groups, which have many of the same attributes as a process, but with important differences. As with tasks, when processes are assembled into groups, or when process groups act as elements of a larger group, the scheduling specification at a given level may be superseded by that of a higher level, or may be used to represent a number of significantly different values. The use and meaning of

```
proc_grp_desc   ::=   Process_group(name) { [pg_attr]* } ;
pg_attr         ::=   Fault_tolerance { ft_spec };
                  |   Process_graph { succ_list } ;
                  |   sched_spec;
                  |   timing_spec;

ft_spec         ::=   alternative_spec
                  |   copies_spec;
                  |   pb_spec;
                  |   voting_spec;
```

Figure 7: Process Group Specification

scheduling specifications for elements of a group is determined solely by the scheduler being used. For example, a deadline for a process may represent an actual deadline when it stands alone, but be a relative deadline or be ignored when the process is part of a group. Figure 7 gives the elements of a process group description.

The specification of process groups serves several purposes, which has the effect of complicating the grammar. However, the basic idea is fairly simple; a process group defines a computation to the system. This means that when processes are grouped, we do not activate or deactivate the processes individually but only as elements of the group. The situation is complicated by the fact that we permit nesting of process groups, to produce arbitrarily complex structures.

A process group has the same scheduling and timing specification sections as a process, for the obvious reason that these apply to any computation whether it is described as a single process or as a process group. The computation as a whole can reasonably specify a deadline, importance, and so on. These are defined in the scheduling specification *sched_spec*, as defined in Figure 5. The computation can also be periodic, and so require parameters given in the *timing_spec*. However, when describing a computation as a group of processes, there are additional properties that can apply.

The most obvious aspect of a process group is the specification of the group structure using the same successor list method as for a task group. The precedence constraints between processes are translated into precedence constraints that hold between tasks in the groups describing the processes involved. Arbitrarily complicated structures can be described since a given process group can be an element of another process group at a higher level.

3.1.4 Fault Tolerance

The most interesting part of a process group description describes its fault tolerance properties. Figure 8 gives four varieties of fault tolerant structures: alternative, copies, primary-backup, and voting. The alternative type is useful for handling *scheduling* faults. The alternative list specifies that different elements of the group represent the computation in differing ways, and the order in which they should be considered. If the system fails to

```
alternative_spec   ::=    Alternate proc_name_list;

copies_spec        ::=    Copies proc_name (min_cp, max_cp);
                     |    Copies proc_name INTNUM;
min_cp             ::=    INTNUM
max_cp             ::=    INTNUM

pb_spec            ::=    primary_proc backup_proc
                     |    backup_proc primary_proc
primary_proc       ::=    Primary proc_name;
backup_proc        ::=    Backup proc_name;

voting_spec        ::=    Voting { voters arbiter };
voters             ::=    Voters proc_name_list;
arbiter            ::=    Arbiters proc_name_list;
```

Figure 8: Fault Tolerance Process Groups

schedule one representation, then the next process on the alternative list can be tried.

The other three types are meant to address *execution* faults in different ways. The *primary-backup* type specifies a pair of processes which are *both* scheduled, but where the primary should be scheduled in such a way that if it completes successfully, the backup can be canceled. If the primary fails, then the schedule constructed will run the backup process by its deadline. This is useful for specifying a preferred version of a computation, the primary, and a minimal version, the backup.

The *copies* type of fault tolerance is appropriate to situations where only one version of a computation is available, but the developer wishes to guard against hardware faults. In that case, the number of copies of the process may be specified. Either an absolute number of copies, or a minimum and maximum can be given. In the latter case, the scheduler has some freedom to choose the number in the context of the current system load. Finally, the *voting* type of fault tolerance specifies the set of voting processes, and the arbiter process that collects the results. Note that the arbiter process could itself be a group.

These constructs are a first approximation of those that will eventually be required as fault tolerance issues are addressed within Spring. The role of the SDL is limited, but vital, since it serves the role of presenting the system with the fault tolerance information desired. Supporting the fault tolerance specified is a responsibility shared by many parts of the system. However, even these constructs present nontrivial issues for process activation and system booting. In doing so, they help make the basic operations of the Spring system more robust and bring it closer to the form that will be required when schedulers handling fault tolerance are implemented.

3.2 Resources

The computations in the system, described as processes and groups of processes, interact with one another through messages, shared memory, and the use of resources. Synchronous messages are exchanged through communication ports, which were discussed in Section 3.1.1. The implications of synchronous communication on program translation and task group structure are discussed in [3]. In this section and the next we consider resources and shared memory segments, which are closely related but separate kinds of objects in the Spring system.

From the Spring scheduler's point of view, a *resource* is simply an abstract object used by tasks in either shared or exclusive mode. Their use of resources implies constraints on what tasks can be scheduled to execute concurrently, and thus constrains the execution plans that can be constructed by the scheduler. From a process's perspective, resources are generally data structures that are shared among some set of processes. However, an obvious exception to this is that the processors on which the processes run are resources for the purpose of scheduling. Recall that Spring uses a distributed memory multiprocessor as its target architecture, in which processes are assigned to particular processors, within whose memory they reside. The assignment of each process to a processor is specified by the system layout, as discussed in Section 3.6. The use of the appropriate processor resource by a task is noted in the run-time data structures constructed during process activation.

From the programming point of view, the resources represent objects in the system to which exclusive access is at least *sometimes* required. Spring-C supports the *with* statement describing the use of a resource, which specifies either exclusive or shared access[3]. Critical sections in a program can thus be represented as exclusive use of a given resource, and the scheduler will enforce the exclusive use constraint. While a resource is used to *represent* a set of shared data structures to which access must be controlled, a shared memory segment is required to *contain* them. Note that this approach works well even in the case of using a resource to represent a set of hardware control and status registers, since they are usually memory mapped. The basic problem, then, is to provide a way to describe the abstract idea of a resource in a way that is appropriate to the requirements of the scheduler, and to describe shared memory support for the data structures comprising resources in a way that is appropriate to the requirements of the SGS and operating system.

Figure 9 gives the grammar for describing a resource. Each resource has a name, and specifies what kind of access is permitted: shared, exclusive, or both exclusive and shared. Note that it is superfluous, from a scheduling point of view, to define a resource unless at least one process uses it in exclusive mode, since otherwise it produces no constraint on scheduling and could be ignored. However, we can also use the resource definition to group shared data structures, exporting them for use by the processes requiring them. As a result, we permit the declaration of a resource which is only used in shared mode.

This is also prudent considering the extent to which software commonly evolves. Introducing a block of code requiring exclusive use of a data structure which was not already represented as a resource would require establishing a resource and then finding all uses of the shared data structure in the code, placing it in a *with* block. It is much simpler to represent sets of shared data structures as resources from the beginning. Resources used only in shared mode can be identified through analysis of the code, during compilation, and the

```
resource_desc   ::=   Resource(name) { [res_attr]* };
res_attr        ::=   Access access_type;
                  |   Segment name;
                  |   export_sym
                  |   Instances INTNUM;
                  |   Mode mode_type;
                  |   Type res_type;

res_type        ::=   Read
                  |   Write
                  |   RW
access_type     ::=   Both
                  |   Exclusive
                  |   Shared
export_sym      ::=   Export name_list;
                  |   Export export_type name_list;
export_type     ::=   Direct
                  |   Indirect
mode_type       ::=   Appl
                  |   Both
                  |   Sys
```

Figure 9: Resource Description

software optimized by eliminating the resource in question from the view of the scheduler. The advantage of this is that when an exclusive use block is introduced, the code only needs to be recompiled, not rewritten.

Each resource also has a type, specifying whether access to it is limited to reading or writing, or if both are permitted. This information can be used by the SGS and operating system to enforce such restrictions. The description specifies a mode for the resource as an aid in checking the final system layout, and as a way of supporting future development. The modes declare whether the resource is used by system or application processes, and provides for the possibility that a resource could be used by both.

The resource description includes a list of the symbol names comprising the resource. These symbols are *exported* from the resource by name, either *directly* or *indirectly*. The latter distinction has extremely important implications for the ability of the SGS to properly predict access time to the shared data, and for the operating system to provide proper support for resources at run time. A data structure can be exported *directly* if all processes using the data structure reference it directly by name, or by address through a pointer that is not shared. A data structure must be exported *indirectly* if a process can access it using a pointer, *which is itself shared*. Direct exportation of a shared data structures places a less stringent constraint on the placement of the shared data structures within the address spaces of the processes using them, so it is important to draw the distinction.

The reason for this arises from the fact that the data structures represented by the resource must be contained within a shared segment that is attached to the address spaces of two or more processes. If the data structures within a resource were exported directly, then each process could map the segment to a different address within its space, and still be able to access the data. However, if the data structure is exported indirectly, then the value of the shared pointer must be valid in the spaces of *every* process using it.

This implies that the segment containing the structure being accessed through the shared pointer must be assigned to the *same* address in all spaces to which it is attached. If the segment containing the indirectly exported data structure was *not* assigned to the same address in all spaces, then the shared pointer's value would be invalid for at least one process. The direct and indirect classification thus helps the SGS in checking for errors during compilation, is required to support the eventual automation of assigning resources to shared segments, and has implications for the the problem of determining access time to shared data, as discussed in Section 2. The classification also has important implications for the properties of the shared segment containing the resource, as discussed in the next section.

3.3 Shared Segments

Shared segments in the Spring system have a number of interesting and unusual properties, in addition to those present in conventional systems. A set of system calls enables processes to create and attached shared segments to their address space at run-time. This interface is currently intended for use only during process initialization, but could in principle be used even when executing under real-time constraints. Such shared segments have no internal structure from the system's point of view, and must be accessed using conventional methods of pointer assignment and manipulation. However, this makes the problem of determining the access time for the structures more difficult, as discussed in Section 2. While the system will eventually be able to handle the the conventional method of access to dynamically created shared segments through pointers, it is also convenient to permit the definition of shared segments at compile time within which data structures are accessed by name.

In the Spring system, shared segments exist as independent objects. They can be defined at compile time using the SDL, and created during system boot, prior to the activation of any processes. Figure 10 shows the SDL grammar for a shared segment description. Segments defined at compile time are said to be *predefined*, since they always exist when processes wishing to attach these segments to their address space are activated. On the other hand, shared segments can also be created during process initialization using the appropriate system calls. In that case, the order in which the processes using the segment are activated can influence the location of the segment within the memory of the system. The use of shared segments by Spring-C programs is discussed further in [3], while the system implementation issues are discussed in [5].

Predefined shared segments are useful in a number of ways including: as a way to standardize access to memory mapped control and status areas for external devices of all kinds, to support system status information made available to application programs, and to ensure unique logical addresses for segments supporting shared data structures that are exported indirectly. Every predefined shared segment has a name which is used at compile time by the SGS, at system initialization time by the system activating a process, and by each process

```
shd_seg_desc   ::=   Shared_seg(name) { [seg_attr]* };

seg_attr       ::=   Code name;
                   | Logical_base HEXNUM;
                   | Matching;
                   | Memory_mapped;
                   | Mode mode_type;
                   | Physical_base HEXNUM;
                   | Predefined;
                   | Resources name_list;
                   | Size HEXNUM;
```

Figure 10: Shared Segment Description

attaching the shared segment to its address space using the shared memory system calls during its initialization.

Segments which are *Memory_mapped* must also specify the *Physical_base* address of the area where the status and control registers are mapped. No physical memory is allocated for these segments from the page pool maintained for each processor within a Spring node. A mode is specified for each segment for the same reasons of consistency and error checking as applied to the resource descriptions given in Figure 9. The *Matching* attribute specifies that the segment must appear at the same logical address in all spaces to which it is attached. This is required to properly support indirectly exported data structures, as discussed in Section 3.2. Since a *Matching* segment is always predefined, *Matching* implies *Predefined.*

The logical base address of a segment can either be specified using the SDL or assigned at system boot time, but the choice is constrained by how processes sharing the segment expect to access the data structures. If the application wishes to access the data structures in the segment by name, then a logical address for the segment must be specified in the SDL description. If the segment has the *Matching* attribute because it contains indirectly exported data structures, then the *Logical_base* is the address at which the segment will appear in each space to which it is attached. This is the only option which is currently supported by the SGS, because of its effect on the problem of determining memory access time.

The *Logical_base* must be specified, and an executable for the segment produced, because an executable image for the process accessing the shared data structures by name cannot be produced unless the logical addresses of those shared variables are known at link time. The Spring system makes this possible by using executable files to represent shared segments. Each shared segment containing symbols referenced by name is described by an executable file whose symbol table gives the logical address of each symbol, as determined by the base address specified when the segment's executable is produced. The *Code* statement specifies the name of the executable file. When the executable for a process using the shared segment is linked, references to the segment's data structures are resolved using the address specified in the segment's executable file.

```
node_desc          ::=   Node(name) { [node_attr]* };
node_attr          ::=   processor_desc
                     |   mem_board_desc

processor_desc     ::=   Processor(name) { [processor_attr]* };
processor_attr     ::=   memory_area_spec
                     |   Use processor_use;
mem_base           ::=   HEXNUM
mem_size           ::=   HEXNUM
processor_use      ::=   Appl;
                     |   IO;
                     |   Sys;
mem_board_desc     ::=   Mem_board(name) { [mem_board_attr]* };
mem_board_attr     ::=   memory_area_spec
memory_area_spec   ::=   Memory_area(mem_board_use, mem_base, mem_size);
mem_board_use      ::=   Data;
                     |   Memory_mapped;
```

Figure 11: Node Structure Grammar

A segment containing only directly exported data structures could specify a *Logical_base* of zero. Then, when the process using it is linked, a base address for the segment can be assigned, and the addresses of the shared structures relocated relative to the assigned base. Thus, each process using this shared segment could attach it to their address space at a different place. However, the modifications to the linker required to support this are not part of the next few steps in the Spring development plan.

If the logical base address is *not* specified, then it will be assigned during process initialization, either by the system or when a shared memory system call is made. A segment's size is always specified, and the set of resources supported by the segment is also listed. Under the current system implementation, each segment must use at least one page of memory. Since this will often be much larger than required for the data structures of a given resource, it is prudent to place more than one resource in a segment. While the method of supporting logical memory may change, obviating the need to group large numbers of shared structures into a single segment, some need to group resources onto segments will probably always exist.

3.4 Node Hardware Description

The portions of the SDL which describe application software have been described in previous sections. This section discusses the portion of the SDL used to described the hardware onto which the software is loaded. The grammar for this section of the SDL is specified in Figure 11. Each node is given a name, and contains boards which are either processor or memory boards. Each processor board is given a name, and the address range

```
network_topology   ::=    Network { [node_connections]* };
node_connections   ::=    Node (name) { connection_list };

connection_list    ::=    connection_spec
                     |    connection_list, connection_spec
connection_spec    ::=    Connection (node_name) { [connection_attr]* } ;
connection_attr    ::=    Latency INTNUM;
                     |    Speed INTNUM;
```

Figure 12: Network Topology Grammar

of the memory associated with it, if any, is defined. Every memory board is also named, and its address range specified. The names given to boards only need to be unique within the node.

Note that the memory board notation is used to describe all memory mapped devices, as well as normal memory. For example, the memory mapped control and sensor registers for a robot would be described as a memory board. For such devices, it is reasonable to permit defining more than one memory area under a single name. Note that in the case of the memory board, the memory area has a use attribute. This distinguishes between memory used for data and that mapping device registers, which gives the SGS information helping it to detect and avoid the error of misusing a memory mapped board by assigning data structures to it.

Finally, note that the node description does not contain a section specifying how the processors within a node are connected. Several descriptions are possible, including one analogous to that given for the network topology in Section 3.5. We currently assume a simple bus connection among all boards in the node, which is why a single base address is sufficient to identify the location of the memory associated with the board. Arbitrarily complex target hardware could be supported by fairly straightforward additions to the grammar to represent interconnections, and their access time characteristics.

3.5 Network Topology

The compilation and downloading of the system and application software only requires a view of the structure of nodes, but if the system is meant to run on a network of nodes it is prudent to permit a description of the connections among them. This information can be of use to the system as it boots each node, since it must establish connections to its companions on the network. Figure 12 gives the grammar for describing the network topology. Each node in the network has a section specifying the names of the nodes to which it is connected, and the properties of the connection.

The current version of the network specification is quite simple, because the current needs of the Spring system are modest. Each connection has a basic *Latency* value associated with it, which gives the time require to begin transmission of a message across the network, and the

```
system_layout       ::=   Layout { [node_layout]* };

node_layout         ::=   Node_layout(name) { [node_layout_attr]* };
node_layout_attr    ::=   processor_layout
                        | mem_board_layout

processor_layout    ::=   Processor_layout(name) { [proc_layout_item]* };
proc_layout_item    ::=   Appl_set { [proc_load_item]* };
                        | System_set { [proc_load_item]* };

mem_board_layout    ::=   Mem_board_layout(name) { [mb_layout_item]* };
mb_layout_item      ::=   Appl_set { [mb_load_item]* };
                        | System_set { [mb_load_item]* };

proc_load_item      ::=   Boot_proc name;
                        | Process_set name_list;
                        | Resource_set name_list;
                        | Shared_seg_set name_list;

mb_load_item        ::=   Resource_set name_list;
                        | Shared_seg_set name_list;
```

Figure 13: System Layout Grammar

Speed attribute, which gives the rate at which information is transmitted, once transmission has begun. These are intended as generically useful attributes, but others will be required to properly describe networks supported by different types of hardware.

3.6 System Layout

The sections of the SDL describing all the objects that can be loaded onto a Spring node have been discussed in previous sections, as have the portions of the SDL describing the node and interconnection architecture. The only part of the SDL left is that used to describe the assignment of the software to various portions of the hardware, which requires specifying the set of items assigned to each processor and memory board within each node. Figure 13 shows the grammar for describing the system layout.

The layout specification reflects two basic ideas: that the application software and system software are described separately, and that each set of software can be described in terms of processes, resources, and shared segments. Memory boards, obviously, can only support resources and shared segments. We list the system and application items for each board separately to help support effective research and development. This is important because

while research in real-time applications may use a standard set of system code, many different versions of the system code may be used for research addressing operating system issues. Specifying the sets separately makes it easier to understand the structure of a particular experiment.

While the application and system code are both described as sets of processes, resources, and shared segments, there are some subtle semantic differences. For example, system and application processes will not necessarily be scheduled in the same way, and so the role of resources in the two sets may differ significantly. The system set for each processor must also identify the booting process (*Boot_proc*), whose executable is loaded first and its execution started. The boot process takes care of properly initializing the board, and establishes a communication channel with the boot processes on the other boards within the node and with the debugger **sbug**. The boot process is then ready to start downloading the rest of the system and application objects, creating and activating them in the boards of the target hardware as specified by the layout specification.

Using separate resource and shared segment lists creates a certain amount of redundance, but we accept it for two reasons. First, the two lists address information used by different parts of the system. The resource list applies to scheduling, since resources represent scheduling constraints, while the segment list specifies some of the executable files loaded on each board. Second, while most resources will be supported by a shared segment, not all will require it, and not all shared segments will support resources. Separate lists let the system retain enough flexibility to handle every situation.

4 SDL Example

The SDL is quite large, and while its function is fairly simple, an illustrative example may make how it is used easier to understand. In this example we assume that the target node has the structure described in Figure 14. The node is given the name *Robot_controller*, and contains five boards; three processor and two memory boards. The *SP* processor board is used for supporting system functions, has $4Mb$ of memory, which is accessible across the system bus at the physical address $0x1000000$. The two application processor boards each have $4Mb$ of memory also accessible from the system bus at the addresses specified.

The *Scramnet* board is a memory board capable of holding data, is $2Mb$ in size, and is visible at the specified bus address. The robot control registers are defined by the *Control_board* memory board description, and occupy the $0x200$ bytes of memory at the specified address. Such information is quite straightforward, even dull, but is clearly required if the SGS, debugger, and other tools are to have a clear view of the target hardware structure. Note how easily the descriptions can be created and modified.

Figure 15 shows the specification for two shared segments and a resource. The shared segment named *Instr_seg* is used only by application processes, and the logical addresses of the data structures it holds are given by the symbol table of the executable file *instr_seg*. The segment has the *Matching* attribute, specifies its *Logical_base*, supports the *Instr_res* resource, and is $8K$ in size because of the page size in the current Spring system implementation. Clearly we would wish to group more resources on the page, if possible, to avoid wasting space.

```
Node(Robot_controller) {
  Processor(SP) {
    Memory_area(Data, 0x1000000, 0x400000);
    Use   Sys;
  };
  Processor(AP_1) {
    Memory_area(Data, 0x1400000, 0x400000);
    Use   Appl;
  };
  Processor(AP_2) {
    Memory_area(Data, 0x1800000, 0x400000);
    Use   Appl;
  };
  Mem_board(Scrammet) {
    Memory_area(Data, 0x14000000, 0x200000);
  };
  Mem_board(Control_board) {
    Memory_area(Memory_mapped, 0xc00000, 0x200);
  };
};
```

Figure 14: Node Structure SDL Example

```
Shared_seg(Instr_seg) {
  Code         instr_seg;
  Mode         Appl;
  Matching;
  Resources    Instr_res;
  Logical_base 0x50000;
  Size         0x2000;
};
```

```
Shared_seg(Robot_cntl_regs) {
  Mode          Appl;
  Physical_base 0xc00000;
  Predefined;
  Size          0x200;
};
```

```
Resource(Instr_res) {
  Access           Exclusive;
  Segment          Instr_seg;
  Export  Indirect Instructions;
  Export  Direct   Instr_p;
  Mode             Appl;
  Type             RW;
};
```

Figure 15: Resource and Shared Segment Examples

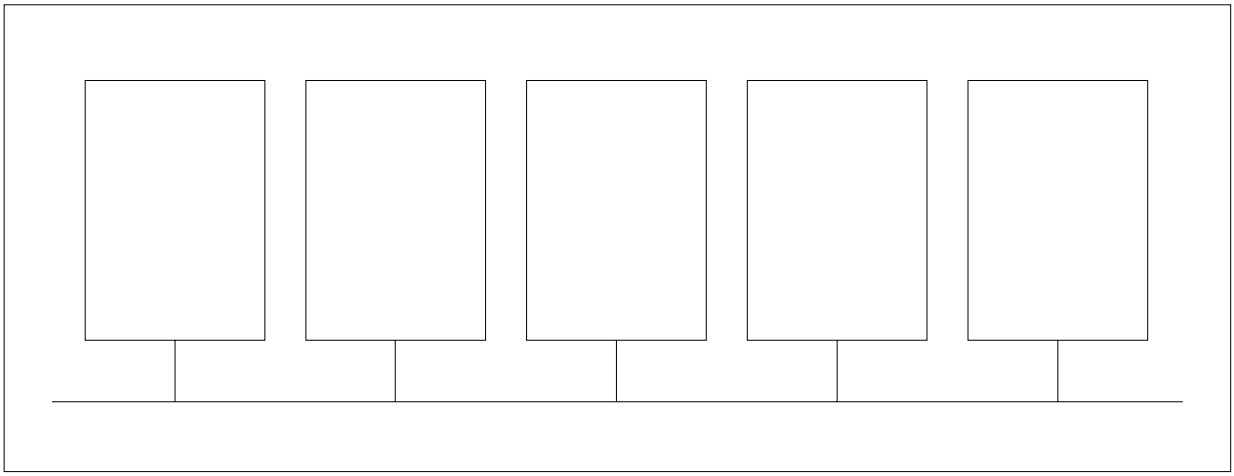


Figure 16: Diagram of Example Hardware

The resource definition gives it a name, and specifies that it is exporting the data structure *Instructions* indirectly, and *Instr_p* directly. Access to the elements of the resource is always exclusive, and the resource is supported by the segment *Instr_seg*. Only application processes use the resource, but it can be both read and written.

In contrast to *Instr_seg*, the shared segment *Robot_cntl_regs* is not associated with any resource. It is assigned a physical base address, since it represents control registers mapped to the specified physical memory addresses. The segment is predefined, and is thus available to processes wishing to attach it to their address space. As it is attached to each process's address space, part of the operation is assigning it a logical address and adjusting the process's memory map accordingly. No resource is associated with it, and no executable file is used to describe its internal structure. The processes using it must thus gain access to its control registers through pointer assignment and manipulation.

The specifications already given establish the structure of hardware on which the application software will run, and the shared segments available for attachment to an application process's address space. Figure 16 illustrates the hardware structure specified by the SDL description of Figure 14. We can now consider three application computations sharing the target system. The first computation is simple; the single process *Reflex* which attaches the *Robot_cntl_regs* segment to its space, and which implements low level reactive control for the robot.

Figure 17 shows the SDL process description for *Reflex* noting that the executable file containing its code is named "reactive", that it uses the *Robot_cntl_regs* shared segment, and is periodic with a period of 25 time units. Its deadline is also 25, relative to the period, meaning that the scheduler is free to plan its execution anywhere within each period. Both the deadline and the process are classified as being *hard*.

```
Process(Reflex) {
   * Exec Spec
   Code          reactive;
   Sharing       Robot_cntl_regs;

   * Timing Spec
   Periodic;
   Period        25;

   * Scheduling Spec
   Deadline      25;
   Deadline_type Hard;
   RT_type       Hard;
};
```

Figure 17: Single Reactive Process Example

```
Process(Planner) {
  * Exec Spec
  Code       planner;
  Import     Instr_p;
  Sharing    Instr_seg;

  * Timing Spec
  Non_Periodic;

  * Scheduling Spec
  Deadline      300;
  Deadline_type Hard;
  RT_type       Hard;
};
```

(a) Planner Process

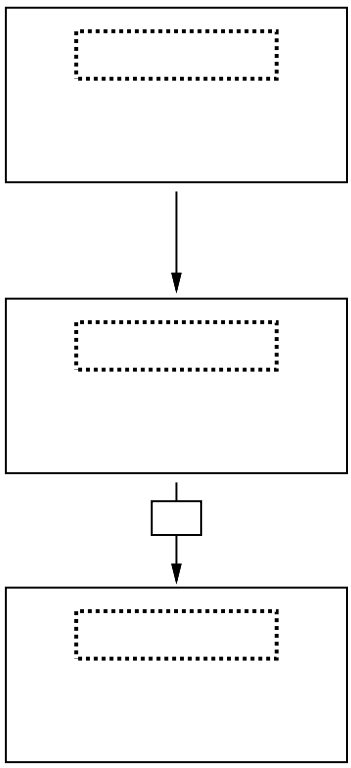
```
Process(Joints) {
  * Exec Spec
  Code       joint_control;
  Import     Instr_p;
  Sharing    Instr_seg,
             Robot_cntl_regs;

  * Timing Spec
  Periodic;
  Period 50;

  * Scheduling Spec
  Deadline      50;
  Deadline_type Hard;
  RT_type       Hard;
};
```

(b) Joints Process

Figure 18: Process Description Example

```
Task_group(Planner) {
  Group_list {
    Begin: T1;
    T1:    T2;
    T2:    (T3 20);
  };
  Task(T1) {
    Non_preemptive;
    W_time   50;
  };
  Task(T2) {
    Non_preemptive;
    Resources (Instr_res Exclusive);
    W_time   40;
  };
  Task(T3) {
    Non_preemptive;
    W_time   80;
  };
};
```

(a) Task Group Description



(b) Task Group Structure

Figure 19: Task Group Example

The other two application computations are a set of two processes that cooperate with one another by exchanging information through the shared data structure represented by the *Instr_res* resource. The two processes obviously share the segment supporting the resource, and one of them also uses the *Robot_cntl_regs* segment. Figure 18 shows the SDL describing the two processes. Figure 18a describes the *Planner* process, whose executable file is named "planner", and which requires access to the shared segment *Instr_seg*. The segment supports the resource exporting *Instructions* indirectly, and *Instr_p* directly. The *Planner* process thus imports *Instr_p*. The planner is nonperiodic, but when invoked *must* be completed within the deadline specified, since the deadline is *hard*.

Figure 18b describes the joint controller process *Joints*, with the file "joint_control" giving the executable code, and which requires access to *Instr_seg*, since it imports *Instr_p*. This process is periodic, and *must* complete by the end of its period. The process also requires access to the control registers of the robot, and so uses the *Robot_cntl_regs* segment.

Each of the processes, when compiled, will be described as a set of tasks. Figure 19(a) gives the task group description of the *Planner* process, while Figure 19(b) illustrates the structure of the group described. It is a simple linear group, with each task having different execution times. The second task uses the resource containing the shared data structure in exclusive mode. The tasks are nonpreemptive, and any deadlines within the group are

```
Layout {
  Node_layout(Robot_controller) {
    Processor_layout(SP) {
      System_set { Boot_proc  sp_boot; };
    };
    Processor_layout(AP_1) {
      Appl_set { Process_set Planner; };
      System_set { Boot_proc  ap_boot; };
    };
    Processor_layout(AP_2) {
      Appl_set {
        Process_set    Joints, Reflex;
        Resource_set   Instr_res;
        Shared_seg_set Instr_seg;
      };
      System_set { Boot_proc  ap_boot; };
    };
    Mem_board_layout(Control_board) {
      Appl_set { Shared_seg_set Robot_cntl_regs; };
    };
  };
};
```

Figure 20: Layout Example

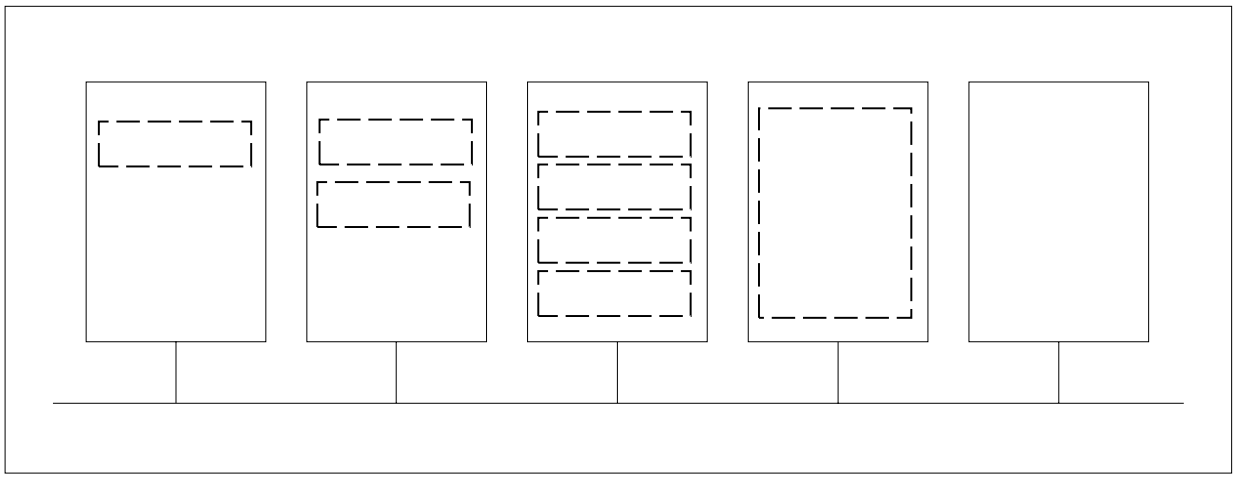


Figure 21: Target Hardware Loaded

left for specification by postprocessing after the executable has been produced, or when the process is activated by a scheduler specific operation. Note the delay associated with the precedence constraint between *T2* and *T3*.

Given the definitions for all the processes, resources, and segments, it is necessary to describe how they are placed in the target hardware. Figure 20 gives a layout specification for the robot control example. It specifies that the SP processor will be loaded only with the executable for the process required to boot it. The *AP_1* application processor supports the *Planner*, while *AP_2* supports the *Joints* and *Reflex* processes as well as the shared segment *Instr_seg* and the resource *Instr_res* it supports. Each of the application processes boots using the same executable image. The memory board *Control_board* obviously supports the shared segment describing it. Figure 21 illustrates the layout specified.

This example has shown how processes, shared segments, and resources can be described, and their arrangement on the system specified. The important point should be clear; that the SDL provides a way to precisely describe the information required to load and run the Spring system. The fact that this information requires a great deal of space is inconvenient when giving examples, but is a direct result of the level of detail being represented. This level of detail is not frivolous, but is that required to support the specification, loading, and running of real-time programs in a predictable manner.

5 The SDL Interface Library

In the Spring programming environment, the SDL information specified by or derived from different source files in the course of compiling the application code is contained in the executable files produced, as well as being accumulated in a single file. This was discussed qualitatively in Section 2 and illustrated in Figure 1, which appears on page 3. This section discusses the specific set of SDL utility routines available for use by tools needing to use, modify, or add to the descriptive information contained in a Spring executable, object, or pure SDL file. The utility routines are contained in the library "libsdl.a", are used by the SDL commands **sdl_cc**, **sdl_merge**, and **sdl_read**, and by the Spring-C compiler **spr_cc**. Structure and procedure templates required by tools wishing to use the utility routines are defined in the "sdl.h" header file.

Figure 22 illustrates the three forms that descriptive information takes in different contexts. The source form is written by developers either as part of Spring-C source files, or as independent specifications. SDL source is parsed by **sdl_cc**, or **spr_cc**, using the SDL parsing engine *sdl_parse* which builds a data structure containing the SDL information in the memory. This data structure holds each category of information as a linked list rooted in the *db_t* structure illustrated in Figure 23.

For example, each process in the system is described by an member of the list rooted by the *process_list* field in the *db_t* structure. As illustrated, each structure describing a process contains several pointers to other structures. One points to the string containing the process name, while three other pointers root linked lists of structures specifying the names of imported data structures, shared segments, and synchronous communication ports used by the process. The process data structure illustrated thus corresponds to the information specified in the SDL grammar for describing processes given in Figure 5 in Section 3.1.1.

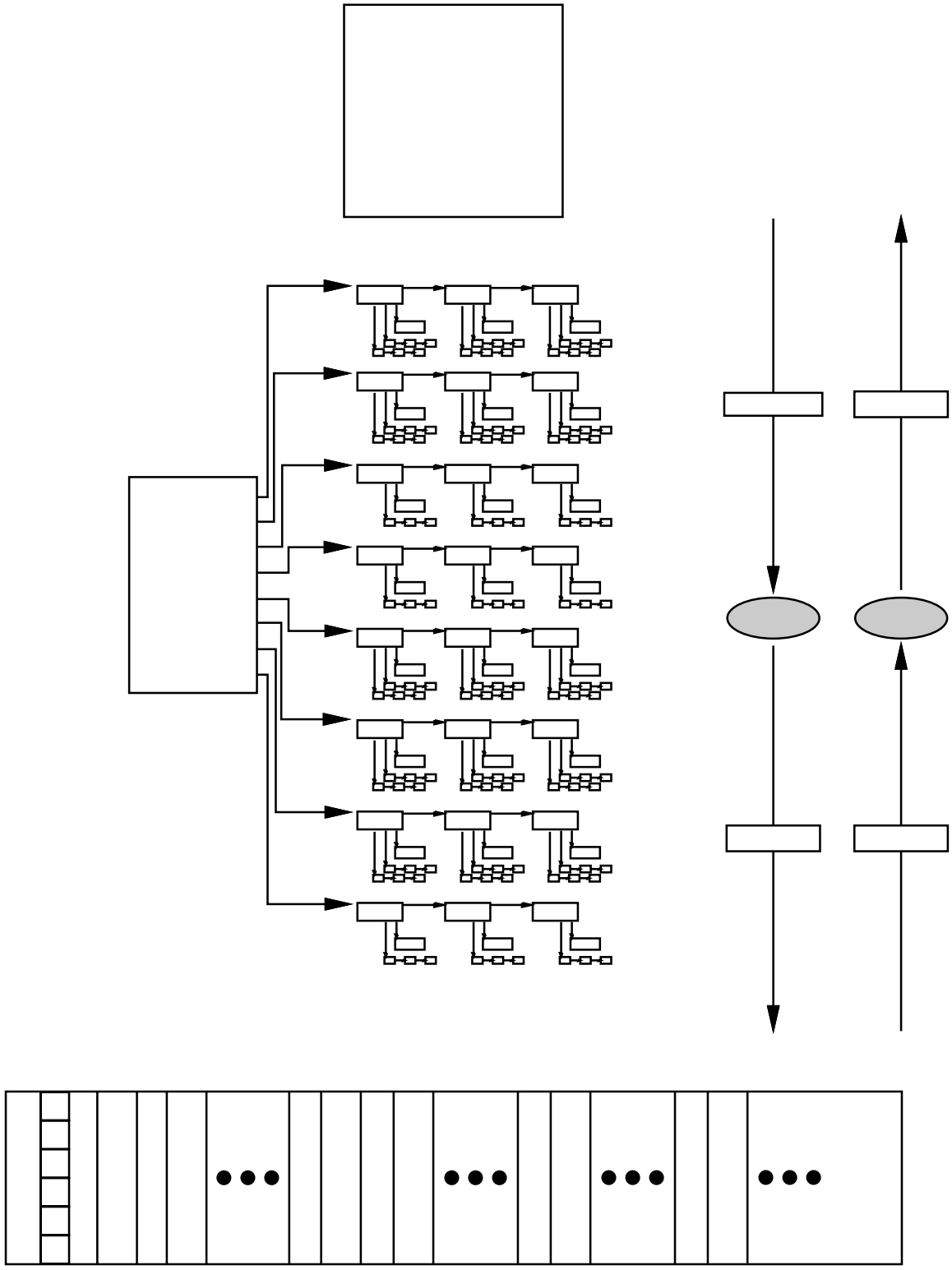


Figure 22: SDL Formats

```
typedef struct db {
    /*
     * the items field is a mask saying what's here.
     * The dir is a set of seek positions that
     * indicate the location of various sections
     * when the information stored in a file.
     */
    int                 items;
    long                dir[DB_NUM_ITEMS];

    /*
     * These are the lists of the various items
     */
    process_list_t      process_list;     /* processes        */
    proc_group_list_t   proc_group;       /* process groups   */
    res_list_t          res_list;         /* resources        */
    seg_list_t          sh_seg_list;      /* shared segments  */
    task_group_list_t   task_group;       /* task groups      */
    node_struct_list_t  node_struct_list; /* target node arch */
    layout_t            layout;           /* software layout  */
    net_top_t           net_top;          /* network topology */
    itg_list_t          itg_list;         /* irred time graph */
} db_t;
```

Figure 23: SDL Data Base Structure Type: db_t

The same approach is taken to the other types of information represented by the SDL and discussed in earlier sections. All tools using, deriving, or modifying the SDL information operate on the lists held by the *db_t* structure. The linked list approach is easy to understand and unlikely to incur any significant performance penalty since the number of items on each list is usually fairly small. However, if SDL descriptions become large enough to require it, as the system evolves, more efficient indexing schemes can easily be added while preserving the basic list structure and thus backward compatibility with existing tools.

Note that the *itg_list* field roots the set of irreducible time graph (ITG) representations of the run-time behavior of each procedure that has been compiled, although there is currently no SDL grammar for specifying ITGs in source files. This section of the SDL, as discussed in Section 1, is used to store the ITGs of each procedure as they are derived during compilation. The ITGs of each procedure are then available during compilation of procedures calling them for substitution into the time graphs of the procedure being compiled. However, if it becomes desirable to create a source grammar for describing ITGs, it is a straightforward matter. One reason a source grammar for ITGs may be needed is to specify the ITGs of Spring system calls.

When a tool using, deriving, or modifying a set of descriptive information is finished,

it needs to write each list of descriptions rooted in the *db_t* structure out to a file in such a way that the original set of lists can be reconstructed. The routine *write_db* supports this operation, taking a pointer to the *db_t* structure as input, and writing the linked list data structures out to memory in "compiled" form, as illustrated in Figure 22. Note that the *db_t* structure is the first thing written to the file. Tools using the compiled form of the SDL as input use the *read_db* routine to read them, and reconstruct the original linked lists in memory. The *read_db* routine uses the first two fields of the *db_t* structure to help access specific sections of the SDL information in compiled form, thus speeding up operations concerned with a specific item.

The *items* field is a bitmask showing which of the linked lists contains information, while the *dir* field is a table of *seek* positions within the file where the information for each linked list begins. When a program calls the *read_db* routine, it can set the *items* field within the *db_t* structure given as an argument, thus specifying which lists should be read. The *read_db* routine then uses this information, and the seek positions specified in the *db_t* structure stored at the beginning of the file to read only those sections requested. It is important to note that the same format for the compiled SDL information is used in a file containing only SDL, and for the SDL section of a Spring executable file. As a result, the *read_db* and *write_db* routines can be used by a tool operating on either type of file.

Finally, the *print_db* routine provides the ability to print descriptive information in readable form. Other interface routines exist to print information in summary form, add items to lists, merge information stored in more than one *db_t* structure, search for descriptive information by name of the object, and to help tools locate the SDL section of Spring executable files.

The rest of this section will present the utility routines in each category. Note, however, that we do not discuss the *sdl_parse* routine since our current view is that **sdl_cc** and *spr_cc* should be the only tools used to compile SDL source, and that all other tools using, producing, and manipulating SDL information should work with the memory and compiled forms.

5.1 Utility Routine Definitions

Programs wishing to use, generate, and manipulate the SDL information can do so easily by using a simple set of utility routines falling into just a few categories. A routine wishing to work with a set of SDL information must allocate a *db_t* structure, and then initialize it:

```
extern void
init_db(db_t *db);
```

The *init_db* routine initializes the *items* field to indicate that the SDL data structure is empty, and initializes every component list as empty as well.

When that is done the tool may wish to read SDL information in compiled form from a pure SDL or Spring executable file. An important consideration at this point is that since Spring is set up as *cross compilation* environment, it is important to know if the file containing SDL information is in *host* or *target* format. In general, all executable files contain SDL information in *target* format, and all non-executable files are in *host* format. All tools

are expected to open the file containing the SDL information and thus obtain an active file pointer for the file. When that is done, the routines:

```
int
is_exec_file(FILE *fp);

void
seek_to_sdl_section(FILE *fp);
```

serve to identify the file as an executable, and to position the file pointer at the beginning of the SDL section respectively. Note that *seek_to_sdl_section* works correctly on both executable and non-executable files. When the file pointer is positioned correctly, the routine reading the SDL information can call one of two routines to perform the operation:

```
int
read_db(FILE *bin_file, db_t *db);

int
read_db_target(FILE *bin_file, db_t *db);
```

Note that the tool using the routines is responsible for identifying the input file type, so that it knows which routine to call. When the read routine returns, the *db_t* structure contains the roots of linked lists holding the SDL information contained in the input file. Note that the tool can also specify what categories of information it desires by initializing the *items* field in the *db_t* structure. When set to zero, as the *init_db* routine does, the items fields requests all available information.

Some routines may read SDL information from more than one source, using separate *db_t* structures for each set of information, but wish to merge the information into a single set. The routine:

```
void
merge_db(db_t *db1, db_t *db2);
```

supports this information, merging the information in *db1* and *db2* together, storing it in *db1*, and leaving *db2* empty.

Searches for objects in the *db_t* structure can be conducted by name in each category. The routine:

```
item_t *
get_db_item_name(db_t *db, int type, char *name);
```

supports this operation, taking a pointer *db* to the *db_t* structure holding the information, the information *type*, and the *name* of the objects as arguments. Note that the constants identifying each category are defined in the "sdl.h" header file.

Tools generating SDL information use the various allocation routines defined in "sdl.h" to create new structures of any given type, and then will fill them in with the descriptions being created. When new information is generated and should be added to the set held by the *db_t* structure, the routine:

```
void
store_db_obj(db_t *db, int type, item_t *obj);
```

supports the operation. The first two arguments are the same as for the *get_db_item_name* routine, while the third is a pointer to the object being added.

When the tool manipulating the SDL information is finished, it may wish to write its results out to a file. It is responsible for positioning the file pointer in the output file correctly, using the *seek_to_sdl_section* routine. When properly positioned, the routines:

```
int
write_db(FILE *bin_file, db_t *db);

int
write_db_target(FILE *bin_file, db_t *db);
```

write the contents of the *db_t* structure out to the file in the host and target formats, respectively.

Finally, whether the information was written out or not, the tool doing the manipulation will often wish to deallocate all the heap storage used to hold the SDL information. The routine:

```
void
dealloc_db(db_t *db);
```

supports this operation, deallocating all structures holding information in the *db_t* structure given as the argument. However, note that individual deallocation routines exist for each object type, and for lists of each object type. Templates for these routines are given in "sdl.h".

While simple, the interface routines discussed here will be sufficient for most of the tools wishing to read, manipulate, and write SDL information. For those with more ambitious plans, the host of routines defined in "sdl.h" are available.

6 Current Status and Conclusion

This document has presented the SDL, shown how it can be used to describe many aspects of a real-time system, how that descriptive information is made available to all parts of the system, and how the information is used to control many aspects of program compilation and execution. This is a moderately prosaic, but vital, aspect of a real-time system. The SDL is designed to play a central role in the Spring system, but this will necessarily emerge as the system implementation matures.

Currently, the SDL tools **sdl_cc**, **sdl_read**, and **sdl_merge** are fully operational, and the SDL interface library "libsdl.a" is available, and being used by several areas of current development in the Spring system. Within the SGS, the SDL parsing engine *sdl_parse* is fully integrated into the Spring-C compiler **spr_cc**, enabling the compiler to take Spring-C source files containing SDL statements as input, and producing object files containing the SDL information in compiled form. However, modifications to the Spring linker **spr_ld** are still

required to properly propagate the SDL information from object files into the executables. That issue will be addressed in turn, but for the moment, it is a simple matter to use *sdl_merge* to accumulate the SDL information in the object files being linked, and **add_sdl** to write the merged information into the newly created executable's SDL section.

The Spring operating system boot operations are currently being extended to use the SDL layout information to control the order in which **sbug** sends shared segment creation and process activation requests to the system, and to use the SDL information contained in the processes' executables when building the scheduler's run-time data structures during process activation. This code is being written so that most of it can also be used on Spring for process activation, and on the host to prepare the Spring Scheduling CoProcessor simulation input.

The current version of the Spring Simulation Testbed uses the previous version of the SDL, which described computations as sets of independent tasks. It will be modified to use the new version, and thus take a process oriented perspective, as those using it require the new abilities. It is important to note that many people are still interested in investigating new approaches to scheduling independent tasks, so the current version is still in active use.

In summary, the SDL is fully implemented, and performing its function as an information exchange medium and basis for integrating the many parts of the Spring system. Development on several parts of the system is proceeding, and uses the SDL information made available by other tools, though full integration will be achieved only after current development is complete. Changes to SDL are easy to make, and will be made in response to problems that arise, and as requirements for new types of information emerge from our implementation experience. When complete the Spring SDL will represent an extremely rich, diverse, and quite detailed specification for the set of information required to write, compile, load, and execute real-time application software.

Acknowledgements

We would like to thank Eric Ludlam and Timothy Farrar for the many hours of work they cheerfully devoted to implementing the SDL language as we had designed it, and for the many helpful suggestions they made during that effort which made the final result more consistent, rational, and robust than it would otherwise have been.

References

[1] Elizabeth Gene. The Spring Simulation Testbed - V2
User's Guide. Technical report, Spring Project Documentation, 1990.

[2] D. Niehaus. Program Representation and Translation for Predictable Real-Time Systems. In *Proceedings of the IEEE Real-Time Systems Symposium*, pages 53–63. IEEE, December 1991.

[3] D. Niehaus. Program Representation and Translation in the Spring System. Technical report, Spring Project - in preparation, 1992.

[4] D. Niehaus, E. Nahum, and J. A. Stankovic. Predictable Real-Time Caching in the Spring System. In *Proceedings of the Eighth IEEE Workshop on Real-Time Operating Systems and Software*, pages 80–87. IEEE, May 1991.

[5] Douglas Niehaus, Chung-Huei Kuan, and Decao Mao. Address Space and Resource Management in the Spring System. Technical report, Spring Project - in preparation, 1992.

[6] K. Ramamritham, J. A. Stankovic, and Perng-Fei Shiah. Efficient Scheduling Algorithms for Real-Time Multiprocessor Systems. *IEEE Transactions on Parallel and Distributed Systems*, 1(2):184–194, April 1990.

[7] J. A. Stankovic and K. Ramamritham. The Spring Kernel: A New Paradigm for Real-Time Systems. *IEEE Software*, 8(3):62–72, May 1991.

A SDL Grammar

System Description Language Top Level (3)

```
config_info ::=   [conf_item]* system_layout [conf_item]* network_topology [conf_item]*
                | [conf_item]* network_topology [conf_item]* system_layout [conf_item]*
conf_item   ::=   node_desc
                | process_desc
                | process_group_desc
                | resource_desc
                | shared_seg_desc
                | task_group_desc
```

Successor List (3.1)

```
succ_list     ::=   succ_begin [succ_item]*
succ_begin    ::=   Begin: prec_list;
succ_item     ::=   name: prec_list;
prec_list     ::=   prec_item
                  | prec_list, prec_item
prec_item     ::=   list_item
                  | (list_item delay_val)
list_item     ::=   proc_group_name
                  | proc_name
                  | (proc_name task_name)
                  | task_name
delay_val     ::=   INTNUM
                  | (Comm_delay port_name_list)
                  | (Comm_delay INTNUM port_name_list)
```

Process Specification (3.1.1)

```
process_desc  ::=   Process(name) { [proc_attr]* } ;
proc_attr     ::=   exec_spec;
                  | timing_spec;
                  | sched_spec;

exec_spec     ::=   Code name
                  | Import name_list;
                  | Sharing name_list;
                  | Sync_ports port_list;

timing_spec   ::=   Non_periodic;
                  | Period INTNUM;
                  | Periodic;
                  | Separation INTNUM;

sched_spec    ::=   Deadline dist_spec;
                  | Deadline_type dln_type;
                  | Importance INTNUM;
                  | Laxity dist_spec;
                  | RT_type rt_type;
port_list     ::=   port_item
                  | port_list, port_item
port_item     ::=   name
                  | (name use_type)
use_type      ::=   Receive
                  | Send
```

Task Group Specification (3.1.2)

```
task_grp_desc   ::=   Task_group(proc_name) { tg_def } ;
tg_def          ::=   group_def [task]+
group_def       ::=   Group_list { succ_list } ;

task            ::=   Task(name) { [task_attr]* } ;
task_attr       ::=   M_time dist_spec;
                  |   Non_Preemptive;
                  |   Preemptive;
                  |   Resources [ru_spec]+;
                  |   sched_spec;
                  |   W_time dist_spec;

ru_spec         ::=   (name [use_attr]+)
use_attr        ::=   Start time_val;
                  |   End time_val;
                  |   Exclusive;
                  |   Sim_prob use_prob excl_prob;
                  |   Shared;
use_prob        ::=   FLOATNUM
excl_prob       ::=   FLOATNUM
```

Process Group Specification (3.1.3)

```
proc_grp_desc   ::=   Process_group(name) { [pg_attr]* } ;
pg_attr         ::=   Fault_tolerance { ft_spec };
                  |   Process_graph { succ_list } ;
                  |   sched_spec;
                  |   timing_spec;

ft_spec         ::=   alternative_spec
                  |   copies_spec;
                  |   pb_spec;
                  |   voting_spec;
```

Fault Tolerance Process Groups (3.1.4)

```
alternative_spec  ::=   Alternate proc_name_list;

copies_spec       ::=   Copies proc_name (min_cp, max_cp);
                    |   Copies proc_name INTNUM;
min_cp            ::=   INTNUM
max_cp            ::=   INTNUM

pb_spec           ::=   primary_proc backup_proc
                    |   backup_proc primary_proc
primary_proc      ::=   Primary proc_name;
backup_proc       ::=   Backup proc_name;

voting_spec       ::=   Voting { voters arbiter };
voters            ::=   Voters proc_name_list;
arbiter           ::=   Arbiters proc_name_list;
```

Resource Description (3.2)

```
resource_desc     ::=   Resource(name) { [res_attr]* };
res_attr          ::=   Access access_type;
                    |   Segment name;
                    |   export_sym
                    |   Instances INTNUM;
                    |   Mode mode_type;
                    |   Type res_type;

res_type          ::=   Read
                    |   Write
                    |   RW
access_type       ::=   Both
                    |   Exclusive
                    |   Shared
export_sym        ::=   Export name_list;
                    |   Export export_type name_list;
export_type       ::=   Direct
                    |   Indirect
mode_type         ::=   Appl
                    |   Both
                    |   Sys
```

Shared Segment Description (3.3)

```
shd_seg_desc  ::=  Shared_seg(name) { [seg_attr]* };

seg_attr      ::=  Code name;
                 |  Logical_base HEXNUM;
                 |  Matching;
                 |  Memory_mapped;
                 |  Mode mode_type;
                 |  Physical_base HEXNUM;
                 |  Predefined;
                 |  Resources name_list;
                 |  Size HEXNUM;
```

Node Structure Grammar (3.4)

```
node_desc          ::=  Node(name) { [node_attr]* };
node_attr          ::=  processor_desc
                      |  mem_board_desc

processor_desc     ::=  Processor(name) { [processor_attr]* };
processor_attr     ::=  memory_area_spec
                      |  Use processor_use;
mem_base           ::=  HEXNUM
mem_size           ::=  HEXNUM
processor_use      ::=  Appl;
                      |  IO;
                      |  Sys;
mem_board_desc     ::=  Mem_board(name) { [mem_board_attr]* };
mem_board_attr     ::=  memory_area_spec
memory_area_spec   ::=  Memory_area(mem_board_use, mem_base, mem_size);
mem_board_use      ::=  Data;
                      |  Memory_mapped;
```

Network Topology Grammar (3.5)

```
network_topology    ::=    Network { [node_connections]* };
node_connections    ::=    Node (name) { connection_list };

connection_list     ::=    connection_spec
                       |   connection_list, connection_spec
connection_spec     ::=    Connection (node_name) { [connection_attr]* } ;
connection_attr     ::=    Latency INTNUM;
                       |   Speed INTNUM;
```

System Layout Grammar (3.6)

```
system_layout       ::=    Layout { [node_layout]* };

node_layout         ::=    Node_layout(name) { [node_layout_attr]* };
node_layout_attr    ::=    processor_layout
                       |   mem_board_layout

processor_layout    ::=    Processor_layout(name) { [proc_layout_item]* };
proc_layout_item    ::=    Appl_set { [proc_load_item]* };
                       |   System_set { [proc_load_item]* };

mem_board_layout    ::=    Mem_board_layout(name) { [mb_layout_item]* };
mb_layout_item      ::=    Appl_set { [mb_load_item]* };
                       |   System_set { [mb_load_item]* };

proc_load_item      ::=    Boot_proc name;
                       |   Process_set name_list;
                       |   Resource_set name_list;
                       |   Shared_seg_set name_list;

mb_load_item        ::=    Resource_set name_list;
                       |   Shared_seg_set name_list;
```

Miscellaneous

dist_spec	::=	(C INTNUM)
	|	(E FLOATNUM)
	|	(E FLOATNUM)(INTNUM)
	|	(G FLOATNUM FLOATNUM)
	|	(G FLOATNUM FLOATNUM)(INTNUM)
	|	(N FLOATNUM FLOATNUM)
	|	(N FLOATNUM FLOATNUM)(INTNUM)
	|	(U INTNUM)
dln_type	::=	Hard
	|	Soft
	|	Non_realtime
name_list	::=	name
	|	name_list ',' name
name	::=	$[a-zA-Z\_]^{+}[a-zA-z0-9\_.]^{*}$
rt_type	::=	Critical
	|	Essential
	|	Non_essential
time_val	::=	INTNUM
INTNUM	::=	$[0-9]^{+}$
	|	$-[0-9]^{+}$
HEXNUM	::=	$0x[0-9a-fA-F]^{+}$
FLOATNUM	::=	$[0-9]^{+}.[0-9]^{+}$
	|	$-[0-9]^{+}.[0-9]^{+}$