A Robotic Pick and Place Assembly Application for the Spring Real-Time System

Carlton Bickford, Marie S. Teo, John A. Stankovic

July 1995

1 Introduction

The Spring real-time system [1] and environment has provided methods for program representation that allow programs to meet the predictability demands of a complex real-time application. These application programs must be predictable in three ways; it must be possible to describe at compile-time, the worst case execution time, the use of shared resources, and any synchronous interprocess communication. To meet these demands, the Spring system makes several tools available to the real-time programmer. The Spring-C language makes a few significant changes to the syntax of ANSI C to ensure program predictability. The Spring System Description Language (SDL) [2] augments Spring-C code with a grammar that is used to describe the resource requirements, timing constraints, and importance levels for each process. And finally, the Spring Software Generation System (SGS) [3] includes a compiler, linker, and loader that will provide a path from Spring-C and SDL source code to an executable on the target hardware executing the Spring kernel.

The primary objective of the work described in this paper is to provide a test case for the Spring real-time system. The test case helps to answer the questions about the completeness and ease of use of Spring's method for program representation. Are Spring tools and methods for program representation adequate for describing real-time applications? What strengths for real-time do these tools add to the application? It also provides a body of application source code, both SDL and Spring-C, that has proved useful in development and testing of the compiler and loader.

The test case is derived from an actual application from industry, much of which has been ported to the Spring system. The original application is a robotic workcell that is currently in use for the assembly of circuit boards. While this is clearly not on the scale of the large complex real-time systems of the future that Spring research is geared toward, it has proved to be a first step for the Spring SGS. It makes extensive use of the Spring mechanisms for shared memory and IPC primitives [4], and thoroughly exercises the use of the SDL to describe process behavior. Section 2 describes the original application in enough detail to allow contrasts to be made with the Spring implementation, which is described in section 3. Section 4 offers a user's evaluation of the Spring interface to the programmer and section 5 provides some observations and lessons learned from the entire experience.

2 The Original Application Using Standard Approach

The assembly of circuit boards typically involves an automated process for placing components. This is usually done by high speed dedicated equipment, easily capable of placing 10,000 components per hour. While this equipment can handle most industry standard component package types, there is often a need for more flexible equipment. A user programmable robotic workcell is often used to place package types that are not compatible with the standard process. In this particular case the robotic workcell is used for large components with heat sinks. The components cannot be handled by standard equipment, and require precise placement that cannot be achieved manually.

2.1 Direct Visual Guidance

The dimensions and tolerances of the packages and the circuit board, combined with assembly quality standards, resulted in a required placement accuracy of ±0.001". The Adept One robot from Adept Technology was used for the application. The robot however, is only capable of ±.003" positional accuracy. This is because the robot controller relies on encoder feedback from each joint to position the end-effector at the desired destination and this results in some positional error. The solution, is an approach that became known as direct visual guidance (DVG).

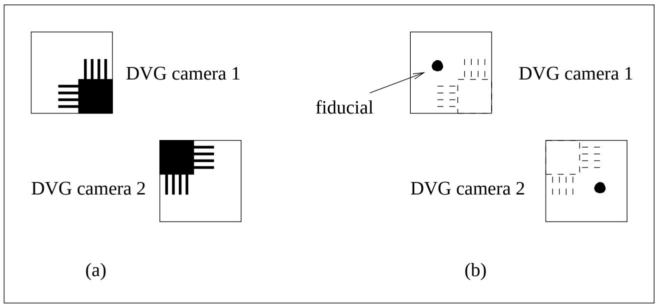

Figure 1: Images acquired by DVG cameras

DVG is accomplished by mounting two cameras on the robot end-effector, in diagonally opposed quadrants. The cameras are used to acquire two different types of images; the corner leads of the component, and a visual target on the circuit board known as a fiducial. After the component is picked from it's feeder the robot will position it over a light source, and acquires an image from each camera (Figure 1a). The corner leads in each field of view allow the position of the component relative to the robot end-effector to be calculated. This computation need only be done once for each placement. The component is then carried to the placement location and is held just above the circuit board footprint on which it will be placed. Each footprint has two fiducials associated with it. An image from each camera is again acquired (Figure 1b), and the position of the robot end-effector relative to the target location on the circuit board is calculated. This computation is repeated, coupled with corrective moves by the robot, until the positional error is acceptable.

2.2 Workcell Configuration

The major components of the robotic workcell consist of the Adept One Robot, the Adept robot controller, and the vision system on a MicroVAX host. There are also many other mechanisms required for the assembly process, e.g., parts feeders, a conveyor, and a tool change rack. The Adept One robot is a direct-drive, 4 axis robot. The robot has a kinematic configuration known as SCARA, (selectively compliant assembly robot arm), that is well suited to planar assembly tasks such as circuit board assembly. Joints 1, 2, and 4 of the Adept One are revolute while joint 3 is prismatic, it moves the end effector in a direction normal to the workspace plane.

The Adept MC robot controller is equipped with three Motorola 10 MHz 68000 processors, all on the same Intel Multibus backplane. The primary processor is used for robot and workcell control and the remaining two processors are dedicated to digital servo control of the robot joints.

The vision system is composed of a Data Translation DT2651 board on the MicroVAX computer. Low-level vision tasks, such as modifying pixel values and selecting frame buffers and lookup tables, are performed by functions available through the IRIS run-time library which is provided by Data Translation along with the DT hardware. Higher level vision routines, (functions that the robot process will call) are linked with the IRIS library. The DVG workcell has four cameras. Camera 0 is mounted on the outer link of the Adept One, and is used for workcell calibration. Cameras 1 and 2 are the DVG cameras. They are rigidly mounted on a fixture which mounts to the robot wrist. Camera 3 is mounted within the robot workspace looking upward, and is not used for the DVG scheme.

2.3 The V+ Control and Programming System

Adept's operating system is V+ [5], formerly known as VAL-II [6]. VAL, the original robot programming system, was developed in 1975 by Unimation. V+ is a multitasking operating system as well as a high level interpretive programming language. For any robotic application, V+ may run two types of user processes. These include one robot control program, and up to six process control (PC) programs.

The robot control program and any process control programs are referred to as user tasks 0 though 6, all of which share the primary CPU with V+ system tasks. V+ uses time slicing and a priority scheme to guarantee that critical processes are given time on the CPU in each 16 ms major cycle. Table 1 shows the default configuration used for V+ scheduling. The major cycle is divided into four slices which must fall on 2 ms increments. The slices are named for the 2 ms increment on which they begin, 0, 5, 6, and 7. A task is assigned to a time slice where it competes for CPU time with other tasks in the slice.

Table 1: V+ scheduling configuration

Time Slice	Slice Duration	Tasks in Slice (default)	Priority
0	10 ms	Robot trajectory generator	255
		User task 0 (robot control)	20
		User task 3 (PC task 3)	15
		User task 4 (PC task 4)	10
		User task 5 (PC task 5)	10
		User task 6 (PC task 6)	10
5	2 ms	None	
6	2 ms	User task 1 (PC task 1)	20
		User task 2 (PC task 2)	15
7	2 ms	Monitor	254
		System device drivers	253

The trajectory generator is the highest priority task and must run to completion every 16 ms. The execution time of the trajectory generator depends on two factors; the type of motion specified (joint-interpolated, straight-line, etc.) [6] and the CPU type. With the 68000 processor, the trajectory generator task could take from 7 ms up to 11 ms. If it does not exceed the time slice then user task 0, the robot control process can execute. This process, due to its priority over other tasks in the slice, will hold the CPU throughout the rest of slice 0 and slice 5 as well (since no tasks are scheduled in it), unless a blocking event occurs, in which case V+ looks for the highest priority unblocked task, first within the slice, and then the previous slice. If no tasks are ready in either, V+ looks forward through the remaining slices. If still no tasks are found, the tasks will be reconsidered again beginning at slice 0. If the trajectory generator task does overflow slice 0, then the task completes in slice 5, and user task 0 from the previous slice would continue execution if ready. Note that user tasks 4, 5, and 6 in slice 0 all have equal priorities. These tasks would be executed round-robin fashion with the least recently executed task getting the available time in each major cycle. Finally, note that by placing user task 1 in time slice 6 with the highest priority in the slice, it is guaranteed to get at least 2 ms out of every 16.

3 The Spring Application

One of the first obstacles to the creation of a robotics test case is the fact that Spring-C is not a robot programming language. Hence, the necessary motion functions, sensor and actuator integration, and spatial descriptions and transformations had to be implemented in Spring-C. The experiment then proceeded by mapping the original application to the Spring system. The basic question is, how can the same application be described in the Spring system, and how well do the Spring methods for program description meet the programmers needs. First some basic assumptions about the application will be made:

1. The original application had the vision process running on separate hardware. This process layout will be paralleled in the Spring application by imposing the constraints that robot and vision processes communicate with each other only through messaging, and that robot processes and vision processes communicate among themselves only through shared memory. These constraints are made to mirror the notion that the vision processes are a remote service offered to the robot routines. Figure 2 shows the conceptual distinction among vision (white ellipses) and robot processes (shaded ellipses). Darkened arrows indicate IPC and light arrows indicate shared memory access.

2. With the exception of the trajectory generator process, the original application has no hard timing constraints. For the Spring implementation, application level timing constraints are imposed. Please see section 3.4.

3. The actual DVG workcell has many functions such as workcell calibration and loading product data files, that have no real-time constraints and would not be useful to implement in the Spring test case. The processes which are implemented (shown in Figure 2), make up a significant amount of code, and are of sufficient complexity to provide a non-trivial test case. The data that would normally be provided by calibration and assembly data is generated in the robot_init process of the test case. The test case workcell is also simplified somewhat and retains only the tooling with which the processes in Figure 2 interact. Of the four original cameras, only the two DVG cameras are used.

4. The original vision routines were responsible for providing the data to determine the positional error of the robot arm relative to the placement location. Since we do not have the vision hardware required, we had to disregard the actual computation involved in the vision system and manipulate just the inputs and outputs to it. Hence, the vision routines are called but are not given any data to work on, instead we simulate the required return values. In other words, the application performs the calls to the vision routines and receives a generated (as opposed to calculated) return value.

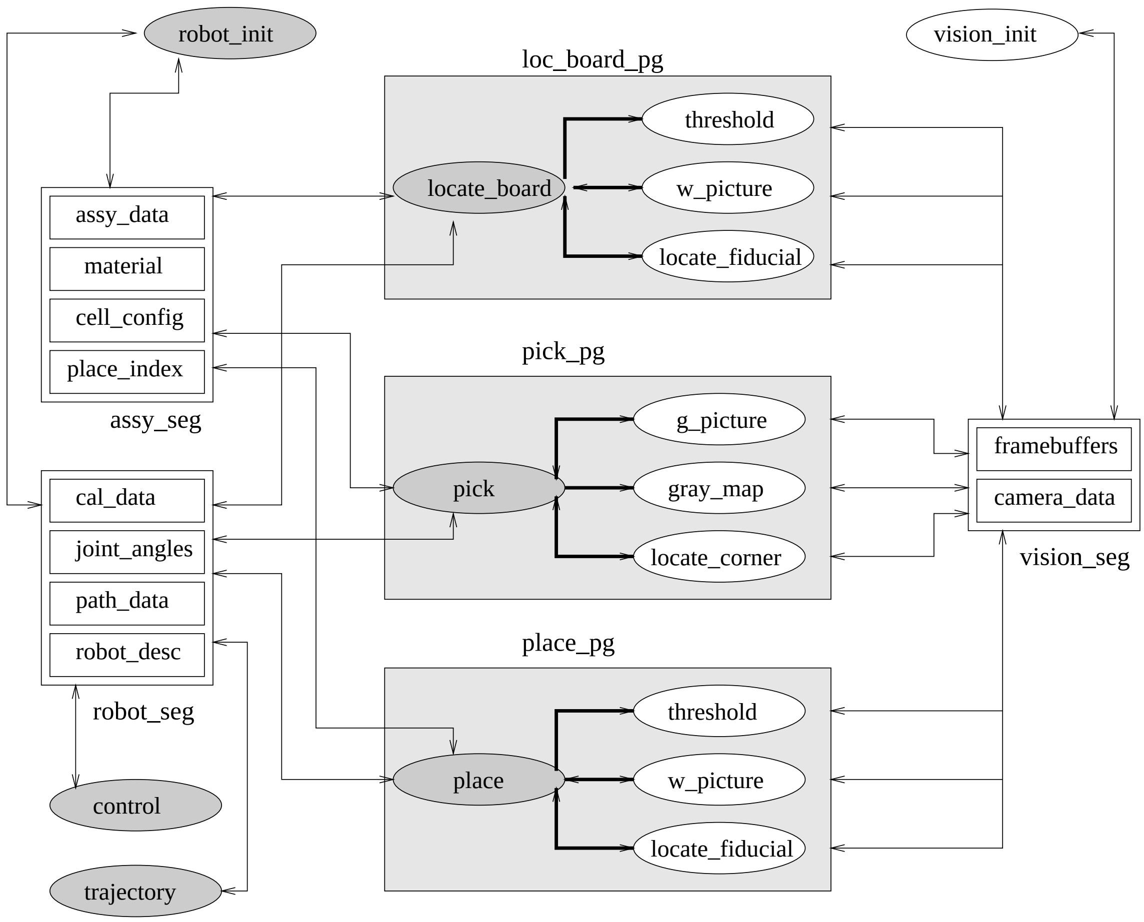

Figure 2: Diagrammatic scheme of the application

5. The original robot controller uses binary outputs and inputs to control actuators and obtain information from sensors. In our application such signals are simulated in software.

6. Originally, placement locations for circuit components are obtained through user input (an input file or CAD/CAM system). However, to remove the need for I/O operations we just randomly generate a set of placement locations at runtime or have the user specify it within a header file.

3.1 Shared Memory

Data structures that are shared among processes are known as resources in the Spring system. Resources are grouped into a shared memory segment that a process must attach using the *shm_attach* system call during process initialization. References in the source code to the resource must then be enclosed in the Spring-C *with* statement. A process may access a resource in either *exclusive* or *shared* mode. This provides a means of implementing critical sections, since the scheduler will enforce the exclusive access. The scheduler need not be made aware of a shared data structure that is not used in exclusive mode by any process. However, as noted in [2], it is useful to describe the data structure as a resource anyway. If at some later point in the software development exclusive access is needed, all references throughout the code will already be properly enclosed in *with* statements.

One of the first steps in developing the application was to decide what data structures would need to be shared among processes, and then to define a reasonable grouping of related data structures into resources. Finally, each resource was assigned to a shared segment. Figure 2 shows the final shared memory design. Shared segments are represented by the large white rectangles, and the resources they contain are shown as the smaller rectangles within them.

A portion of the source code for the segment shared among the vision processes, *vision_seg*, is shown in Figure 3 as an example. Shared segments are represented in Spring using executable files. By doing so, the logical addresses of the symbols are made available at link time to the executable image of any process that references the symbol. A walk-through of the example will demonstrate how shared segments are used in the application, and will demonstrate the particular features of the SDL that are used to describe these segments to the Spring system.

The data type *v_cam_t* holds calibration data and default settings for a particular camera that the vision processes will need. The shared segment contains an instance of this data type for each of the two cameras, *v_cam1* and *v_cam2*. It is convenient to manage all of the cameras in the application with an array of pointers to these camera descriptions, i.e. *cam_array*, as shown. Therefore *v_cam1* and *v_cam2* will be referenced using a pointer that is shared among processes. The data structures *v_cam1*, *v_cam2*, and *cam_array* are collectively referred to as a resource which has been given the name *camera_data*. The SDL **Access** statement specifies that this resource is always accessed by a process in shared mode, and **Segment** gives the name of the segment containing the resource. Because *v_cam1* and *v_cam2* will be referenced indirectly through a pointer by more than one process they are exported indirectly by the resource. This information is needed during compilation to determine the memory access time for the reference. It also places restrictions on the way the segment containing the resource can be mapped to the address space of each process to which it is attached. The reasons for this will be discussed shortly. The symbol *cam_array* may be exported directly, since it will always be referenced by name. **Mode** simply states that this segment is used by an application rather than by the system, and **Type** specifies that a process may both read from and write to this resource.

The beginning portion of the *vision_seg.c* file is the SDL description of the shared segment itself. The name *vision_seg* is used as an identifier to the compiler and linker, to the system as it

```
/* vision_seg.c */

SDL {
    Shared_seg(vision_seg){
        Code            vision_seg;;
        Mode            Appl;
        Matching;
        Predefined;
        Resources       camera_data,
                        frame_buffers;
        Logical_base    0x7C000;
        Size            0x2000;
};

Resource(camera_data){
    Access          Shared;
    Segment         vision_seg;
    Export Indirect v_cam1, v_cam2;
    Export Direct   cam_array;
    Mode            Appl;
    Type            RW;
};

Resource(frame_buffers){
            •
            •
            •
```

Figure 3: vision_seg.c

activates processes, and to any process that must use the name to attach the segment. The SDL **Code** statement gives the name of the executable file containing the symbol declarations. The **Matching** statement indicates that this shared segment must be mapped to the same location in the logical address space of every process that attaches this segment. This is because of the use of the shared pointers discussed earlier. The **Logical_base** gives the location in the logical address space of the process, at which the segment will be attached. It provides the matching location needed to support the indirect references, and also provides addresses of the symbols referenced by name. Although Spring provides system calls that enable a process to create shared segments during process activation, all shared segments used in this application are predefined (i.e. they are created at compile time and are available for processes to attach during activation).

3.2 Process Descriptions

The original workcell implemented the vision routines with a single process which would search through a master list of functions to provide a function pointer used to execute the requested function. This approach would be inefficient for us, since the Spring SGS would produce a run-time representation of the original monolithic process that would include the worst case execution time and resource use of all functions in the master list. Clearly, a finer granularity at the programming level is needed to allow the Spring system to produce a more efficient run-time task representation. This has been accomplished by representing the functions in the original master list as processes. These are shown in the right half of Figure 2.

Spring-C programming convention requires that each process have two entry points. See Figure 4. One designated *proc_init* that will be executed only once at process initialization, and the other designated *proc_exec*, that will be executed at each process episode. The *proc_init* section is used to

```
/* threshold_init.c */

#include<sys/ipc.h>
extern port_id_t Threshold_port_id;

int proc_init()
{
    int first;
   /* create a port, owned by this process, to accept requests
    * from the processes locate_board and place
    */
    name_t     recv_port_name="threshold";
    int rv;

    rv=ipc_port_create(&recv_port_name,
                        1,
                        semantic_type_sync;
                        queue_policy_earliest_deadline,
                        overflow_policy_drop_arrival,
                        &Threshold_port_id);

     /* attach shared memory
      */

    if(shm_attach("vision_seg",1,&first)==0) {
        return 0;
    }
    return 1;
}
```

Figure 4: threshold_init.c

perform such system tasks as attaching shared segments and setting up synchronous communication ports. The *proc_exec* section implements the desired computation.

The SDL source language provides for a process specification which allows the programmer to describe the behavior of a process, in terms of its resource use, timing constraints, importance level, and synchronous communication with other processes. The threshold process specification is shown in Figure 5. The execution specification portion states that the source code is contained in the file *threshold.c*, the symbol *cam_array*, which is contained in the shared segment *vision_seg*, is referenced, and the process receives messages on a synchronous communication port named threshold. The SDL process specification also allows for a description of timing and scheduling requirements. However, since the threshold process will be contained in a process group, these requirements will be specified for the entire group rather than at the individual process level.

```
/* threshold.c */

SDL{
    Process(threshold){
        * Exec spec
            Code          threshold;
            Import        cam_array;
            Sharing       vision_seg;
            Sync_ports    (threshold Receive);
        * Timing spec
        * Scheduling spec
        };
    };
        •
        •
        •
 proc_exec() {
        •
        •
        •
```

Figure 5: threshold.c

Discussion so far has centered around vision processes, but as shown in Figure 2 there are also four robot processes: *robot_init*, *locate_board*, *pick*, and *place*. These are not ported directly from the V+ code of the original application. The functionality of the original workcell has been preserved through the implementation of these Spring processes, while at the same time a process architecture has been imposed that is better suited to the generation of a predictable run-time representation. As with the vision processes, each of the robot processes has an SDL description (not shown due to space limitations).

3.3 IPC

The Spring kernel provides primitives for both synchronous and asynchronous messaging between processes. All interprocess communication in the workcell application is synchronous. This is because all IPC in the application is among robot and vision processes that, as a group, implement a computation. Since there are precedence constraints among the processes in the group, the communication must be synchronized. These process groups will be discussed in more detail later, but as an example consider the group *loc_board_pg* in Figure 2.

The three vision processes must be run in the order: *threshold*, *g_picture*, *locate_fiducial*. Furthermore, the *g_picture* process may not begin until the *locate_board* process has positioned the robot arm such that the camera is positioned over a fiducial. The Spring system enforces these precedence requirements, and also avoids unpredictable blocking, by decomposing the process group into the run-time task representation discussed in [8]. A synchronous send or receive call in the source code is a scheduling point, which forces a task boundary in the run-time representation. This allows the non-preemptable tasks to be scheduled such that when a synchronous receive is executed, the message will be available.

```
/* locate_board.c */
            ⋮
/* send a message to the vision system
  * to set a binary threshold
  */
requestp=(request_t*)&request_buffer;
requestp->cameras=CAM1;
requestp->lo_threshold=board.threshold;
}

sync_send("threshold",Threshold_port_id,
               &request_buffer);
            ⋮
```

```
/* threshold.c */
            ⋮
sync_receive("threshold",Threshold_port_id,
                   &request_buffer);

requestp=(request_t*)&request_buffer;
threshold=request->lo_threshold;
camera=requestp->cameras;
            ⋮
```

Figure 6: Synchronously communicating processes

While much of this translation method is hidden to the programmer, it has an implication that should be considered when applying synchronous IPC. That is, synchronous send and receive calls result in scheduling overhead and precedence constraints, since each call forces task boundaries at the entry and exit of the call. Furthermore, a send call has a communication delay associated with it. Consider for example the two communicating processes *locate_board* and *threshold*, shown in Figure 6. The *sync_send* and *sync_receive* calls in these processes will result in the task group shown in Figure 7.

$T1$ is a task whose ending boundary is the entry point of the *sync_send* call. $T2$ is a task whose starting boundary is the exit point of the *sync_send* call. $T1'$ and $T2'$ are similar tasks whose boundaries are defined by the entry and exit points of the *sync_receive* call. The arrows in Figure 7 indicate the precedence constraints that are imposed. There is no precedence constraint on $T1'$, it may be scheduled to execute before or after $T1$. There are however, two precedence constraints on $T2'$. It will not be scheduled to execute until after both $T1'$ and $T1$ have executed. Furthermore, the scheduler will insert a gap equal to communication delay after the completion of $T1$, which represents the minimum amount of time that must elapse before $T2'$ may begin. The value for communication delay is provided to the scheduler by the real-time network service [4].

The application makes substantial use of IPC, as can be seen from the following example of the synchronous communication that must take place among the processes in the process group *place_pg*:

1. The place process makes *sync_send* calls to set the threshold for each of the two DVG cameras.

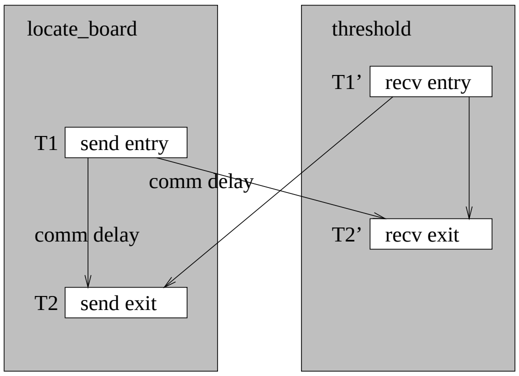

Figure 7: Decomposition of synchronously communication processes

2. For each of the two cameras, the place process makes a *sync_send* call to *w_picture* to acquire a windowed image of the fiducial, and then makes a *sync_receive* call to find out which frame buffer the image has been placed in.

3. For each of the two frame buffers, the place process makes a *sync_send* call to *locate_fiducial* and then performs a *sync_receive* call to acquire the coordinates of the fiducial center.

3.4 Process Groups

A computation may either be implemented as a single process, or as process group. Consider the following computations:

- Pick the next component from the feeder.
- Position the component over the light box.
- Take a gray level picture with each DVG camera.
- Perform gray level mapping to improve the quality of each image.
- Use the images to find corners of the component
- Pass the coordinates back to the requesting process.
- Use the corner coordinates to define a component- to-wrist transform.

This entire computation is implemented by the four cooperating processes: *pick*, *g_picture*, *gray_map*, and *locate_corner*. Together they form the process group *pick_pg*. To the Spring system, the significance of a process group is that all processes in the group are activated and deactivated at the same time. For the programmer, the process group is extremely useful since it allows timing constraints and importance levels to be applied to the entire computation. This is demonstrated by the SDL description of the pick process group shown in Figure 8.

The **Process_graph** section indicates that there are no precedence constraints among the four processes, they may begin execution in any order. The synchronous communication will ensure that the computation proceeds as designed. The timing specification indicates that the entire process group is **periodic** and must run to completion every 500 time units. The scheduling specification

```
/* pick_pg.c */

SDL {
    Process_group(pick_pg) {
        Code pick_pg;
        Process_graph {
            Begin: pick, g_picture, gray_map, locate_corner;
        };

        * Timing spec

        Periodic;
        Period 500; * in 10 ms units

        * Scheduling spec

        Deadline 500;
        Deadline_type Hard;
        RT_type Critical;
    };
};
```

Figure 8: pick_pg.c

gives a **deadline** of 500 as well. Thus if the process group requires less than 500 units for worst case execution, the scheduler may arrange the tasks that make up the group anywhere within the period. (Provided of course, that precedence constraints and communication delays among tasks are observed). The **real-time type** of the process group is specified as *critical.* In the Spring system this means that group must have an a priori guarantee that it will finish on time, and all needed resources must be reserved. The **deadline type** is *hard*, meaning that a negative value is imparted to the system if the computation should miss its deadline.

Each of the three process groups shown in Figure 2, (*loc_board_pg*, *pick_pg*, and *place_pg*) have similar SDL descriptions. Since the three process groups represent the primary purpose for which the workcell was designed, it seems reasonable to designate their type as critical, and require that their execution be guaranteed statically at load time. It should be noted however, that the Spring system allows an alternative design for the application. The three process groups could be designated essential with hard deadlines, and an on-line guarantee would be obtained dynamically the first time the process group was invoked. Since the process groups are periodic, their execution would be guaranteed as long as they remain in the system.

Although the workcell application describes only one robot, it is designed such that the second robot may be implemented by simply making a duplicate copy of the entire application. (Of course names of processes and process groups would need to be modified to ensure unique names.) Referring to Figure 2, the only element that would not be duplicated is the shared memory segment, *assy_seg*. This segment would be shared among both robots, to cooperatively assemble the circuit board. To avoid collisions the robots will alternate in their occupation of the board fixture and feeder areas. This is accomplished by partitioning the assembly process into three major computations (process groups) of roughly equal execution time and specifying timing and scheduling constraints for each, as shown in the following table.

Table 2: Possible schedule for a two-robot workcell

Elapsed time units	0	500	1000	1500	...
Process group executed by Robot 1	loc_board_pg	pick_pg	place_pg	pick_pg	...
Process group executed by Robot 2	pick_pg	place_pg	pick_pg	place_pg	...

4 A Users Evaluation

The interface that the Spring system provides to the programmer is evaluated here at two levels. From a high level, the ability to design an application for Spring is discussed. Of particular interest are the methods that Spring provides to ensure that application is predictable, schedulable, and flexible. At a lower level, the Spring-C programming language is discussed, emphasizing the users experience with the constructs that have been added to ANSI C.

4.1 Application Design

The end-to-end scheduling offered by the Spring system is a powerful and useful method for the programmer to apply timing constraints to a computation that is composed of many tasks. Each of the three process groups described in the application are quite complex. They each include numerous function calls, involving both robot motion and vision processing. They are each composed of multiple processes that engage in synchronous communication and share data structures among themselves. Yet despite the inherent complexity of each process group, there is a need to abstract each as a computation to which overall constraints may be applied.

Managing complexity is a concern in the development of any large software project and this condition is only made worse when computations must be real-time as well. There are several qualities of the Spring programming environment that work well together to manage this complexity while ensuring that the real-time application is predictable. These qualities are enumerated below.

1. Some help in managing complexity is provided by the C programming language, of which Spring-C is an extension. Many features of C which are used throughout the application are not available in V+ or similar robot programming languages. The modularity of C, it's preprocessor, bit operations, conditional compilation, and type definition capabilities are all used throughout the application. These helped to produce code that is much more concise and readable than an equivalent application in V+.

2. The ability to describe a computation as a process or a process group, and the ability to nest groups in arbitrarily complex arrangements (i.e. groups of process groups) is a useful abstraction. When this is combined with the ability to impose timing constraints on the entire group, it becomes a powerful tool for real-time applications.

3. Creating the detailed SDL descriptions required for shared segments, processes, and process groups is a price that the developer pays for predictable program behavior. However, the SDL descriptions proved to be very helpful in the initial high level design of the application. They provided a formal design specification of the process architecture and resource use that was referred to repeatedly throughout development.

4. The synchronization of shared resources and messages among processes could potentially be an extremely tedious programming task, especially when tight deadlines are imposed. Fortunately,

the with, sync_send, and sync_receive constructs in Spring-C effectively hide this detail from the programmer and place the burden of synchronization on the SGS.

4.2 The Programming Language

Spring-C is a modification of ANSI C that eliminates any source of unpredictable behavior in programs. Syntax for loop bounds and recursion depth for recursive procedures is provided, thereby eliminating any unbounded execution. Furthermore, all places where process execution is suspended are made visible to the SGS. Processes in Spring do not suspend, they are decomposed into non-preemptable tasks. Thus, any statement that would normally cause a process to block is known as a scheduling point and forms a task boundary. There are three types of statements in Spring-C that cause scheduling points: *with*, *delay*, and the *sync_send* and *sync_receive* statements. The final change to ANSI C is the elimination of the *goto* statement.

There was no need for any recursive procedures in the application, so the depth bound was not exercised. Loop bounds of course, were used extensively. It is typically a simple matter to specify a minimum and maximum bound for the loop. The only shortcoming is that the bounds are probably much too large in most cases, but the programmer must err on the conservative side. A lower value could be used for an upper bound, and the Spring system will notify the user if a loop bound violation occurs.

Moving on to the statements that force scheduling points, the Spring-C delay statement proved to be essential to the application. There are numerous cases where a robot process will use a *sync_send* call to request that a vision process acquire an image. This request cannot be made however, until the robot has positioned the camera at the appropriate location. The fact that a motion command such as *move(pict_loc)* has been executed does not mean that the robot has arrived at the desired location by the next program statement. V+ provides commands such as *BREAK* and *WAIT* that can be used to synchronize robot motion with other computations. In the Spring application, *delay(int_constant)* serves this purpose nicely. The parameter for the delay statement must be supplied by the user and may have to be fine-tuned during testing. Finally, the *sync_send* and *sync_receive* statements have already been the topic of much discussion, and their use is demonstrated throughout the application.

The above is simply an example of what the programmer is continually reminded of while developing a real-time application. If programs must exhibit predictable behavior, then many of the conventional techniques used in non-real-time programming must be discarded.

5 Observations and Lessons Learned

The experience of designing and implementing this application has taught us a few lessons that are important for future real-time programmers of Spring to bear in mind:

1. In creating the design for the application, the user must know how the use of shared resources and synchronous communication is going to determine the overall run-time representation. An understanding of the target hardware is essential to determine the process layout and the resource layout. An understanding of the translation process is also required. The SGS environment should not be a black box between the user and the completed executable. This is not so much a disadvantage, since a real-time application and its execution environment should be well-understood.

For instance, in this application, we originally thought that synchronous communication would be the significant factor affecting the resulting run-time representation. However, as it turns out, this application is more affected by the use of shared resources and this was due to the fact that we constrained vision and robotic processes to communicate among themselves only through shared resources.

2. The user should analyze the resulting task group patterns in order to understand the behavior of the executable at run-time. The SGS provides the **spr-objdump** tool to view the task group representation in a readable form. At this stage, the user is provided with information that allows making intelligent changes or optimizing the application.

   For instance, in this application, we analyzed the task groups and discovered that explicit delays and synchronous communication account for only a small number of suspension points and that most of the 685 tasks of the compiled application were due to use of the *with* statement within the three processes, *locate_board*, *place*, and *pick*. We decided to review the code to discover if the huge number of tasks was inherently necessary.

3. Within the code the user should pay careful attention to the use of statements that cause suspension points. In particular, how resource use is specified is important since placement of *with* statements can greatly affect resulting task group patterns.

   Again, in this application, we discovered that there were many sections of code in which we could move *with* statements to higher granularities (e.g., one *with* surrounding a larger block of code, rather than several within the the same block) to make the decomposition more efficient. This reduced the total number of tasks to 147, a number which is more reflective of the complexity of the application.

6 Summary and Status

The test case has shown that the SGS, the Spring-C and SDL languages, and elements of the Spring kernel, all work together to provide a reasonable application programming interface. In summary, we reiterate that the application now has (or has the potential for) the added following qualities:

1. Predictability:
   The application is now predictable. The executables contain all the information needed to predictably execute the application. This information is also accessible to the user at compilation time, through the SGS tools (in future there might be some automated compiler feedback during the compilation, for now the user can use **spr-objdump**).

2. Guarantees:
   Looking back on the Spring application described in section 3 it may be noted that most processes and process groups are designated critical. This is clearly not a very challenging environment for the Spring scheduler since if each specified period is sufficiently long, a static schedule will be created and will not change as the workcell operates. However, the strength of the Spring system becomes apparent as new essential or non-essential processes are introduced. Suppose a video display driver is responsible for displaying the current image of the fiducial or component corner on a monitor for the benefit of the workcell operator. The video display driver would be a non-essential process with a hard deadline that would be invoked each time a camera acquired an image. This process and many others like it could be added to the workcell as needed with no concern for the critical processes already in place since their execution episodes and required resources are *guaranteed*.

3. Flexibility:
   The application has greater flexibility. Throughout design and coding, and after, the user is greatly aided by being able to specify timing constraints at many levels, especially at the level of end-to-end scheduling. Making changes to these specifications is easy. For instance, it is easy to change the application to handle two robots (please see 3.4) or to lengthen the period for *locate_board*, *pick* and *place* in order to "slow" the robot down. The application is then able to tolerate updates, changes in the environment (to some extent changes in the hardware) without much rewriting of the code.

4. Fault tolerance:
   Although we do not, the application can make use of the specifications for fault-tolerance provided through the SDL.

This has by no means been an exhaustive test case. There are many features of the Spring system that have not been used, but on the other hand a single application rarely makes use of all services provided by an operating system. Some of the areas that the test case did not cover are listed here. Spring kernel support for fault tolerance and the SDL provisions for describing fault tolerance were not utilized. Also the workload produced by the application is not particularly challenging. It does however, establish a core of critical processes, to which essential and unessential tasks may be added.

In order to demonstrate the application, we wrote an animated graphical display of the robot arm. Instead of an actual robot controller, the joint angles are read by the animator code which then updates the robot display. The compilation and linking of the application has been successfully completed, and it is currently being tested and debugged.

7 Acknowledgments

This paper and the design of the application has benefited from many discussions with Doug Niehaus. Thanks go to Michael Pasieka and Gary Wallace for their help with the Spring SGS. Ken Ward, Michael Kou, and Doug Young created the DVG workcell at Digital Equipment Corporation in 1991.

References

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

[2] Niehaus, D., Stankovic, J. and Ramamritham, K. "The Spring Description Language," Spring Project Documentation, University of Massachusetts, Amherst, MA, May 1992.

[3] Niehaus, D., and Kuan, C.H. "Spring Software Generation System," Spring Project Documentation, University of Massachusetts, Amherst, MA, June 1990.

[4] Nahum, E., Ramamritham, K. and Stankovic, J. "Real-Time Interprocess Communication in the Spring Kernel," Spring Project Documentation, University of Massachusetts, Amherst, MA, May 1992.

[5] Adept Technology Inc. V+ Reference Guide (version 8.0). San Jose, CA, 1988.

[6] Shimano, B., Clifford, G., Spalding, C. and Smith, P. "VAL-II: A Robot Programming System Incorporating Real-Time and Supervisory Control," *Proceedings of SME Robots 8*, June 1984.

[7] Craig, J. *Introduction to Robotics*, Second Edition, Addison Wesley, Reading, MA, 1989.

[8] Niehaus, D., "Program Representation and Translation for Predictable Real-Time Systems," *Proceedings of the IEEE Real-Time Systems Symposium*, pp53-63, 1991.

[9] Pau, L.F., *Computer Vision for Electronics Manufacturing*, Plenum Press, New York, 1990.