Addendum to: A Robotics Application for the Spring Real-Time System

Marie Sie-Een Teo

July 1995

1 Introduction

This work completes the work done by Carlton Bickford, the goal of which was to create a test application for the Spring real-time system that exercises the methods and tools of Spring and provides an evaluation of the system from a practical standpoint. Bickford took an actual application from industry, redesigned it for Spring, ported and wrote the code based on the original application. We continued his work and completed the code, added a graphical animation, debugged and tested the application.

Spring is a predictable real-time system designed for complex hard real-time systems. In order to provide program level predictability, the Spring system incorporates the Spring-C programming language, the System Description Language (SDL) and and the Spring Software Generation System (SGS). Spring-C is an extension of ANSI-C with support for real-time and SDL, which is integrated into Spring-C, is a grammar for describing information required by the system at all stages of compilation, loading, and execution. Finally, the SGS includes a compiler, linker and loader that produce the executables for Spring.

Bickford's original question was: Are the Spring tools and methods for program representation adequate for describing actual real-time applications? Bickford's conclusion was affirmative for the stages of program design, coding and compilation. In completing the project, we retain the original question as well as adding to it: Are the 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? What lessons did we learn from the entire exercise?

2 The Robotic Application

The Spring Robotics application is based on an actual industry application because we are concerned with the utility and completeness of the Spring System tools and methods. The original application is for a visually guided robot arm that performs automated assembly of circuit boards. The Spring application is as faithful to the original as possible given that we had to accommodate the lack of the actual hardware required for the application. Still, the application is a non-trivial exercise of the various Spring tools.

3 The Spring Robotic Application

3.1 The Workcell

Please see [1] for a diagram and specifications of the robot. The configuration of the workcell can be programmer-specified (see section 7.1). This includes the placement of the tool rack, the feeder and the board, and their dimensions. Note, in the real application this data is fed into the application by the operator or from a CAD/CAM system. The code allows for two component types, each component type has a corresponding gripping tool, a feeder type and feeder location. However, in the current version we use only one feeder and one component.

3.2 Rewriting to Spring-C

Most of the issues involved in the rewriting work has been detailed in [1]. The resulting code in Spring-C and SDL implements the same application but is in predictable run-time representation form. We summarize the main issues that were involved in the rewriting.

- Robotics applications require functions for calculating the robot's position and specifying motion. In the original application, such routines were written in the robotic programming language V+, and were translated to Spring-C.
- 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.

- In the original application, the vision processes run on separate hardware. In order to reflect this the application is conceptually divided into processes involved with the vision system, and processes that have to do with the robot and the workcell. To enforce this division, any communication between vision processes and robotic processes must be done through message passing. Among vision processes and among robotic processes communication occurs only through shared memory.

3.3 Simulating the Hardware

In order to demonstrate the application, we provide 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 original application had a vision system provided by a separate processor board and a run-time library. The vision routines are 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 substance of the vision system, manipulating just the input and outputs to it. 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.

In Figure 1 we show a diagram similar to Figure 9 in [1]. The vision routines are shown shaded to indicate that they are empty stub routines. Also, the vision routines declare data that is never used, which is indicated in gray. More specifically, the vision routines can be divided into three categories:

- *w_picture* and *g_picture*: These functions obtain windowed and gray-scale images respectively. As shown, they call the vision stub routines and are set to be always successful.

- *threshold* and *gray_map*: These functions perform the necessary preprocessing on the image data; threshold sets a binary threshold for each image that gray_map uses to perform gray level mapping. Again, these functions call the vision routines and are always successful.

- *locate_fiducial* and *locate_corner*: These functions are more complicated. They call further functions to locate the fiducial and corner coordinates within the image data. We bypass calling those routines and instead just generate a random value within a constant range of the center

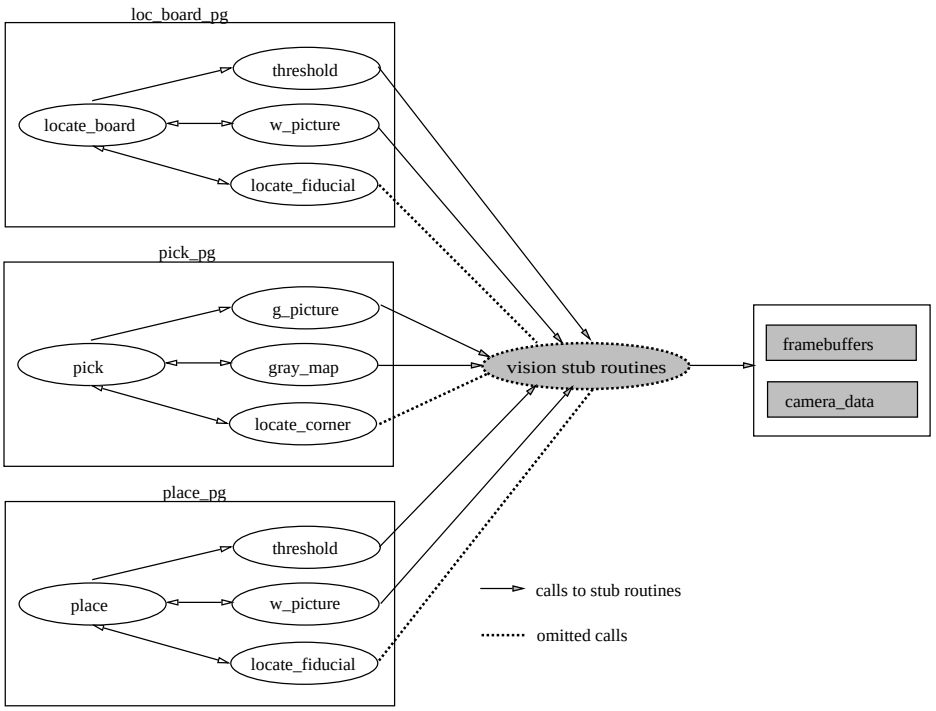

Figure 1: Use of vision routines

of the image and assume this for the value of the fiducial and corner coordinates.

Lastly, although ideally we should obtain placement locations from the user, an input file or CAD/CAM system, the Spring system does not currently have I/O operations implemented and hence we just randomly generate a set of placement locations at runtime or have the user specify it within a header file (see section 7.1).

3.4 Timing constraints

In the original application only one process has hard timing constraints: the process which does trajectory generation. The Spring implementation however allows the programmer to specify timing constraints for both individual processes and groups of communicating processes. First, we list the four main phases of the application :

1. Initiation:
   This phase is non-real time since all that is involved is the initialization of the robot and vision data structures according to the workcell configuration data. We also generate the set of placements.

2. Calibration:
   To translate board locations to robot world coordinates, we need the transform function that describes the board with reference to the robot base. The board has fiducials to aid in this. After moving the robot over to the expected board fiducial location, we can calculate the required transform function.

3. Picking:
   The robot picks next component to be placed from the appropriate feeder, using the appropriate tool.

4. Placing:
   The robot moves to the placement location and takes images of the footprint fiducials. Then the positional error is calculated and the robot readjusts its position. Currently we assume one correction step is enough, we may change this to loop until error is within a certain range.

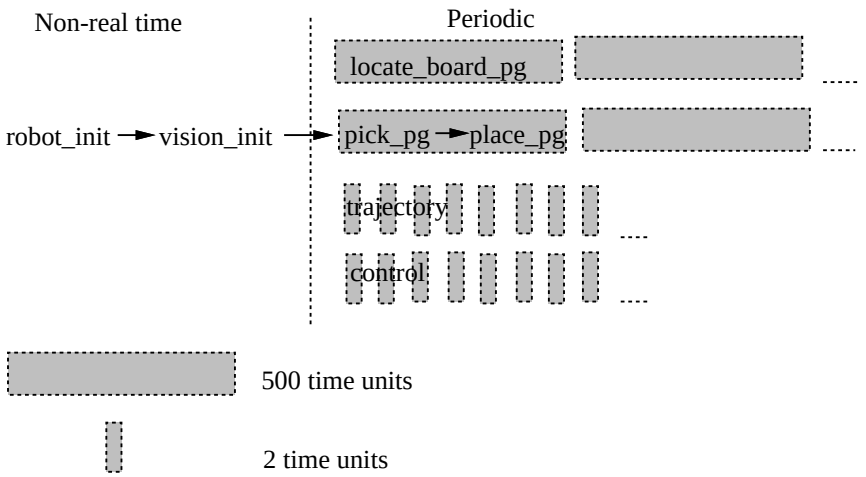

Figure 2: Timing constraints

Here, we reiterate the four phases in the context of their timing constraints, which are specified using SDL (see [1], section 4.4 for a code example). As shown in Figure 2, the initialization is taken care of by *robot_init* and *vision_init*. This is done in non-realtime. Next, calibration is done by the *locate_board_pg* process group which has a period of 500 time units. The picking and placing by the *pick_pg* and *place_pg* process groups have the same periodicity of 500 time units, but with the added precedence constraint, that pick_pg executes before place_pg. Finally, the two processes trajectory and control both have periods of 2 time units.

4 Compilation

The flow from compilation to execution of the application is shown in Figure 3. As each Spring-C file is compiled by **spr_cc**, the behavioral description of each procedure is derived, and included in the resulting object file, and added to *full.db*. The compiler reads this accumulated information each time it runs. When all the application code is compiled, the loader **spr_ld** creates the executables for each process. The SDL information contained in each object file and in *full.db* is used to resolve all links. A complete behavioral description is included in the final executable.

SDL files containing description of the application and the system are compiled by **sdl_cc** into machine readable files containing the descriptive information. All the files are then merged by **sdl_merge** tool to produce a

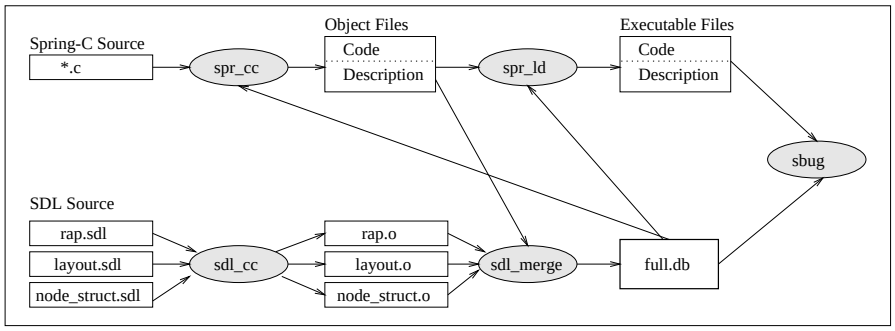

Figure 3: Compilation Information Flow

cumulative description of the application in the file *full.db*.

The executable files and *full.db* provide all the information required to the debugger **sbug** to download the system and application code onto the Spring node as described by the SDL layout description and begin execution.

We have shown where the information specified in SDL is used at different stages of compilation, and where it is finally stored to be used during execution. This information is essentially part of the application and will be used in debugging as will be seen later.

4.1 Program Translation

We have shown the information flow and tools involved in the compilation process. In this section we briefly describe the actual translation process, from the source code and SDL to the executable representation. It is necessary for the user (application programmer) to understand this process, at least at the conceptual level, in order to understand and analyze the resulting executable.

The end result of compilation is an executable for each process, containing the SDL task group description of the process's behavior, which represents all possible execution paths for the process. This behavior is described by a group of non-preemptable episodes of executions called **tasks** which have their own resource requirements, WCET and timing constraints.

Translating the process into such tasks is the key step and is done automatically by the compiler. We provide a very brief overview of the entire process, please see [3] for a full description.

The original **gcc** compiler builds a graph of **RTL** nodes as its intermedi-

ate representation. The Spring compiler uses an enhanced RTL to preserve information on the structure of the code and any suspension points within the code (structural information describes sequences of the source code's block structure while suspension points are places in the code where the execution of the program must or may be suspended). From the enhanced RTL graph, a *time graph* is built which retains the same information but additionally provides WCET timing information for each RTL node. This *time graph* is then reduced to an *irreducible time graph*, a minimal linear time graph of time nodes and suspension points that represents the process behavior. Finally, a task group is built from the irreducible time graph. This task group describes all possible paths through the code. Scheduling all the tasks in the task group in their correct order satisfies scheduling the process.

4.2 Translation of the Application

The following table shows the processes involved in this application and the size of their task groups :

Process	Number of Tasks
control	3
g_picture	7
gray_map	4
locate_board	35
locate_corner	7
locate_fiducial	7
pick	15
place	31
robot_init	13
threshold	4
trajectory	7
vision_init	5
w_picture	9

A process in Spring is set up to execute until the process terminates or until it encounters a suspension point within the process. Suspension points occur when there are explicit delays, resource use and synchronous communications (the corresponding C statements are `delay`, `with`, `sync_receive` and `sync_send`. Therefore, suspension points define the boundaries between episodes of uninterrupted execution and each such episode is a task. Consequently, the task group representation of a process is dependent on the frequency and placement of suspension points within the code.

For example, we show another table of the the main processes and the size of their task groups, before the code was "optimized". We explain what we mean by "optimize" in this case. For a given process, a larger number of tasks in its task group increases the likelihood that a schedule will be found for the whole process since there is greater flexibility. However, this is a trade-off since too many tasks are inefficient if the overhead costs is not justified by the actual computation time (i.e., task computation time is small compared to overhead for each task). Since the code as it was first compiled, had a total of 685 tasks (mostly due to the three processes listed in the table below), a number which seemed too large for the application, we analyzed the task group patterns to see if the number of tasks was necessarily required by the processes or if it was due to how the code was written. If the latter applied, then we could consider how to reduce the number of tasks, making the application more efficient. We also had hardware limits on the kernel memory space to consider and satisfy.

Process	Number of Tasks
locate_board	111
pick	295
place	199

In order to understand the above numbers, consider that a process that has no suspension points will have a task group with a single task in it. The simplest task in our application, *control* has 3 tasks in it arising from one suspension point. All the other processes not listed had task group sizes below 20. We analyzed the code to explain the task group patterns we were seeing.

The relevant fact in understanding the high number of tasks is that this application makes heavy use of shared resources (as shown in Figure 9 of [1]). Explicit delays and synchronous communication accounts for only a small number of the suspension points, e.g. the processes *locate_board*, *pick* and *place* make exactly 10 synchronous communication calls within their code. The large majority of suspension points occur because of the use of shared resources. Using the **spr-objdump** tool, which allows us to view the task group representation in a readable form, we traced the process decompositions, noting where and why tasks were formed. Then, carefully preserving the semantics, we moved or combined suspension points through simple changes to the code to reduce the total number of tasks to 147. This substantially reduced total is more reflective of the complexity of the application.

The following figures are simple examples demonstrating the the type of changes we made to the code. The first figure demonstrates combining the use

of resources and moving the resource specification up to a higher granularity. In the following example function **F1()**, the statement **R1** uses the shared resource **A**, and **R2** uses the shared resource **B**. All other statements are not suspension points. If the programmer places the `with` at the lowest granularity, surrounding the statements **R1** and **R2**, the function decomposes into 5 tasks. If instead, we combine the resource use, making no distinction between **R1**, **R2** and **S3**, merely stating that within that block of code **A** and **B** will be accessed, then the function decomposes into 3 tasks.

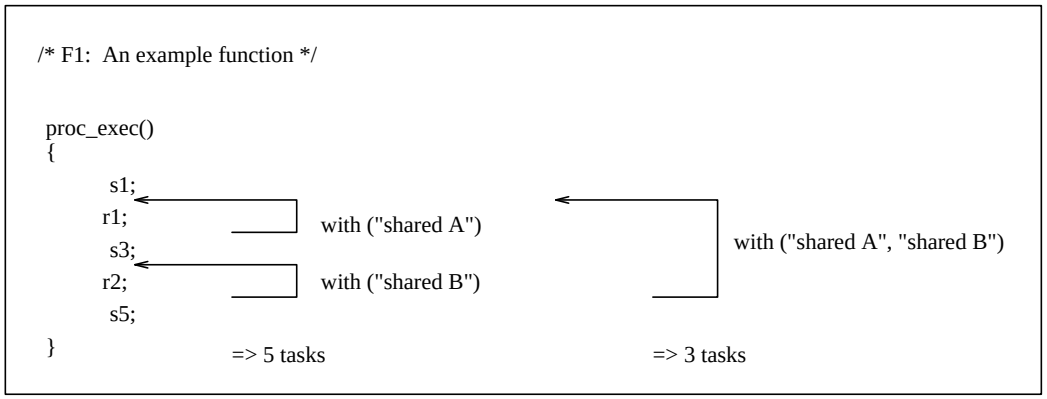

Figure 4: Combining tasks

The next figure shows that we can move the resource use specification to an even higher granularity level by moving the `with` statement out of the function **F1()** entirely, and placing it outside the call to the function. This is efficient when there are multiple calls to **F1()** within the same block of code. Before the move, the procedure calling **F1()** would have 11 tasks for that section of code. After the move, there would be 3 tasks.

Again, we chose to make these changes in order to find a better decomposition for our application, one that was still schedulable and with a reasonable number of tasks. We could imagine that similar changes could be made to optimize the performance of the code. Once the application is translated and executing, such performance tuning can be done.

5 Execution and Performance

The application is awaiting debugging and execution. This section should be completed when that is done.

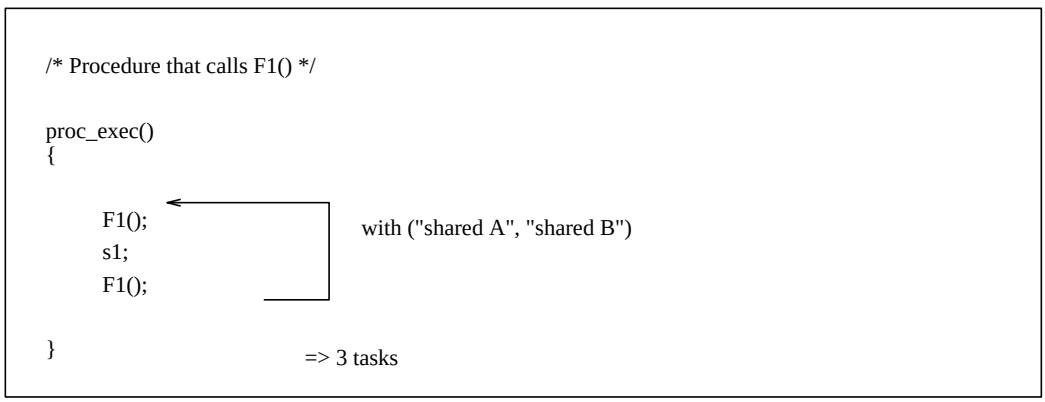

Figure 5: Moving suspension points

6 Evaluation

Essentially, we have demonstrated some of Spring's abilities by taking an industry application and re-writing it for the Spring environment. We have described how Spring-C and SDL is used to make the application predictable while adhering to the overall design of the original application. This shows, as concluded by Bickford, that the Spring system provides a reasonable application programming interface. (Please see [1] for Bickford's evaluation.)

In summary, the application now has (or has the potential for) the 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. 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 Bickford's report) 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.

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

6.1 Lessons Learned

Bickford's evaluation focused on the stages of design and coding and the programming interface provided by the Spring environment. Having completed the debugging and testing stages, we can now make further observations based on all the stages.

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. More is required of the user in this environment. 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. So, the SGS environment is less of a black box between the user and the completed executable. This is not so much a disadvantage, since a real-time application should be well-understood.

2. During coding, the use of statements that cause suspension points within the code, must be considered. In particular, resource use is important since placement of `with` statements can greatly affect resulting task group patterns. The user should aim for a decomposition that is schedulable yet not inefficient.

3. The user should perform the last step of analyzing the resulting task group patterns in order to understand executable behavior. At this stage as well, the user is provided with information that might require making changes or optimizing the application.

7 Source Code

This section is intended as a user and maintenance guide for the test application, we have not included details of the Spring environment.

The application name is the **Spring Robotic Application**. Currently it resides in */usr/users3/spring/kernel_apcode/one_robot/teo*. To compile use **make**. The current configuration is for **RTL3** with **one AP**. Be sure to

change the *Imakefile* when changing nodes. As required, the configuration is described in sdl in `layout.sdl` and `node_struct.sdl`.

The events to trigger execution of the application (there is no main procedure) are set in an event file called *full.evt*. Currently, the full.evt describes one whole period of execution with non-real-time initialization *startup*, one instance of the three periodic process groups, *locate_board*, *pick* and *place* and multiple instances of the *motion* process group.

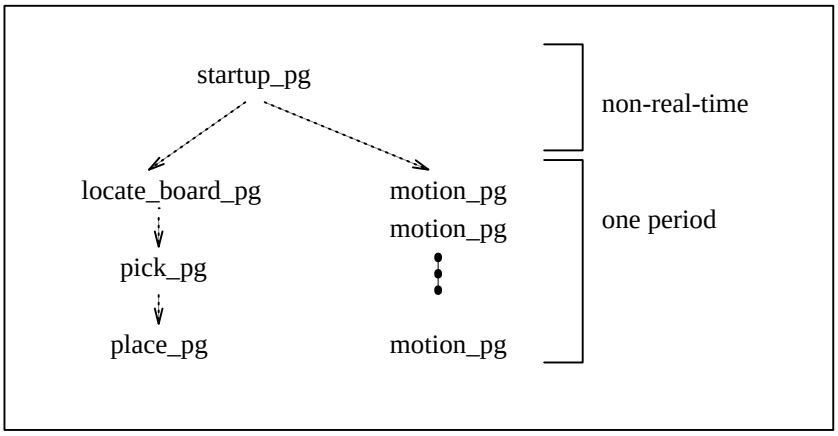

Figure 6: Execution flow

7.1 Content of Modules

The non-real-time processes *robot_init* and *vision_init* initialize the shared resources for the robot, workcell and vision system. They are in the modules `robot_init.c` and vision_init.c, respectively. Their precedence order is specified in `startup_pg.c`.

All shared resources of to do with the robot are declared in `robot_seg.c`, those for the workcell are in `assy_seg.c` and those for the vision system are in `vising_seg.c`.

The *motion_pg* consists of the two processes trajectory and control and they reside in `trajectory.c` and `control.c`.

The three processes *pick*, *place* and *locate_board* are contained within the modules `pick.c`, place.c and locate_board.c. They are the main processes within the `locate_board_pg.c`, *pick_pg*, and *place_pg* and process groups (see `loc_board_pg.c` , `pick_pg.c` and `place_pg.c`). All other processes are synchronized with them (using synchronous IPC). These other processes

are in `gray_map.c`, `g_picture.c`, `locate_corner.c`, `locate_fiducial.c`, `threshold.c` and `w_picture.c`.

The modules, `corner.c`, `circle.c` and `connect.c` contain the vision functions that were originally intended to be called from within the functions in `locate_corner.c` and `locate_fiducial.c`. Currently, we omits the calls to those functions since there is no data to process. The modules `iris.c` and `framebuf.c` contain mostly empty stub routines.

Configurable specifications are found within the following header files:

- assy.h:
  the maximum number of placements per board
  predefined placement coordinates or use random
  number of tools
  feeder style to use
  dimensions and coordinates of toolrack
  dimensions and coordinates of board

- robot.h:
  number of joints of robot
  default speed
  height of the end effector
  dimensions for top view of robot
  distances and rates used for estimating delay times

- vision.h:
  range of allowed error for DVG error displacement

7.2 Animation Code

The graphical simulation is done by **animate**, which resides in the same directory, with the source files under *animate_src* . Within the application, the *control* process periodically changes the robot arm position, according to new joint angles calculated by the motion functions. Within *control*, we log these joint angles to a file. This file is the input file read by **animate**, which takes the angles, recalculates the robot arm positions and replays the motion of the robot arm.

Figure 7 shows the workcell display of the animation. The robot arm is currently positioned over the board and is in the process of placing a component. Figure 8 shows the corresponding lateral view of the board, with the third link of the robot arm descending to place the component.

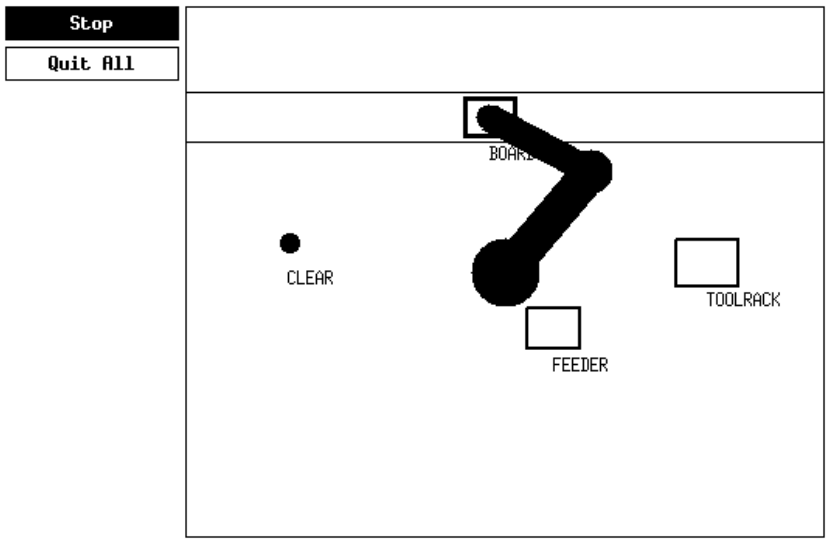

Figure 7: Workcell Display

Placement coordinates are listed, with filled placements highlighted, so we can see that the current component being placed is the second on the list, and the first component has been placed.

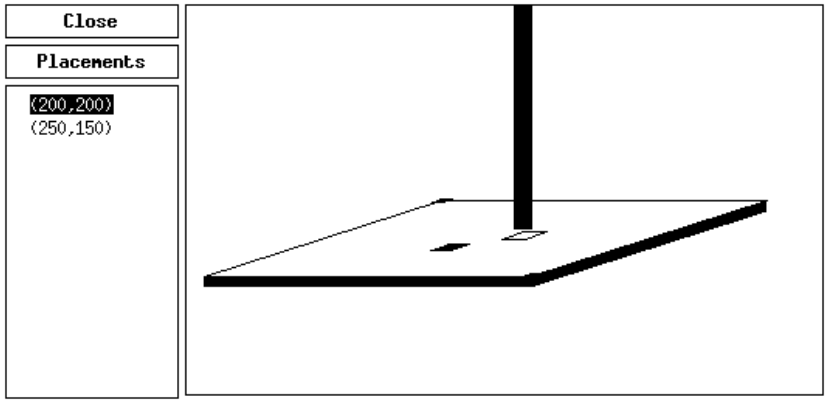

Figure 8: Board Display

The modules `xdraw.c` and `xmain.c` make up **animate**. The configurable options are within the header files:

- xcolors.h:
  color settings

- xsettings.h:
  dimensions for display windows
  scaling factors for the displays
  size and width of display lines and objects

References

[1] Bickford, C., "A Robotics Application for the Spring Real-Time System," Spring Project Documentation, University of Massachusetts, Amherst, MA, Dec. 1993

[2] 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

[3] Niehaus, D., "Program Representation and Translation for Predictable Real-Time Systems," Ph.D. Dissertation, University of Massachusetts, Amherst, MA, Feb. 1994

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

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