Spring Kernel: Added Features (Making Open Spring)

G. Zlokapa G. Wallace S. Kasera

December 1993

University of Massachusetts at Amherst
Computer Science Department

A number of new features have been added to the original Spring Kernel to support the CARTS Demo, where robotics, vision and artificial intelligence processes have to perform in a predictable and synchronized manner to accomplish *time dependent operations*. Starting from the basic idea, the emulation of the flexible manufacturing assembly line, we intend to demonstrate that an operating system such as the Spring Kernel is necessary and very suitable in more complex and dynamic applications of the future. In these applications the external environment is ever changing and the system is required to perform in a timely and predictable manner. It is at least very impractical, if not impossible, to design such a system in a traditional way: a specialized and dedicated system that performs specific tasks. A slight change in the application requirements or in the external environment would require redesign, evaluation, and extensive testing of the entire system.

The rest of the document is organized in sections that talk about individual features and the motivation for their design as well as the current implementation status. Through the document, the term *Original Spring* will indicate the Spring Kernel implementation prior to April 1993. The term *New Spring* will indicate the current implementation that handles tasks with precedence constraints.

0.1 Time Sanity

The basic assumption of the hard real-time system is that the timing characteristics of every hard real-time task will not be violated. If the timing characteristics get violated, it is necessary to detect them and to provide a mechanism for graceful degradation.

In *Original Spring*, the timing violation is detected at garbage collection time. When the task that was scheduled to complete is to be purged out of the system task table (STT), the scheduler checks for timing errors, and if the tasks status is not changed to DONE, the scheduler simply halts all application processors. This error handling schema has several disadvantages. The delay between the occurrence of the timing error and the moment the error is handled depends on the garbage collection frequency that for many reasons can be intolerably long. Additionally, an abrupt halting of all application processors in the middle of their executions is very simple but potentially very dangerous action. For example, if a robot arm is commanded to move forward when the scheduler halts the system, the robot arm would never receive the command to stop moving forward that could result in a wild spin, a spill of a dangerous chemical, or expensive hardware damage.

In *New Spring*, we recognized these disadvantages and implemented more extensive timing error handlers. First, the actual timing error detection is moved from the garbage collection routine to the level of individual application processors. Whenever an application starts a task execution, a dispatcher activates a countdown timer using a task's WCET. If the task finishes early or on time, the timer is reset. However, if the time out occurs, the timing violation is detected and the guilty processor executes the timer ISR. The timer ISR sets the global timing error flag visible to all dispatchers, and it raises its own timing fault flag.

This action has several advantages over the original solution. The timing error is served at the earliest possible time. The faulty processor is detected, providing easier system debugging. Raising the globally visible flag allows all other dispatchers, in the same multiprocessor box, to stop dispatching of new tasks and, thus, preventing a snowball effect that can be caused by the original timing error. Note that by stopping the dispatching of *new tasks* rather then simply halting the entire system, the chance of leaving system in inconsistent and hazardous state is minimized.

Additionally, an application specific process of graceful degradation can easily be implemented by used and invoked by the scheduler. The scheduler checks for any system errors, and if any, it can invoke a user written error handling routine. For example, a user can indicate which tasks to remove or replace with alternative tasks, creating a more robust and fault tolerant system. Further help for troubleshooting is provided by collection of some statistical information at each dispatcher. Within each task control block a dispatcher keeps track of *dispatch in* time (the time from start timer until the end of the change context routine) and *dispatch out* time (the time from user task exit until the timer is reset). This data is used for debugging to detect exactly which part, the system overhead or user given WCET, is responsible for occurred timing error.

Two more timing errors are detectable in *New Spring*. First, scheduler timing error caused by the longer than estimated scheduling overhead is detectable. If the scheduler runs longer than estimated, a newly created STT is not announced to prevent the damage that can be caused by the faulty schedule. Note that catching and handling this timing error properly is important and that ignoring it at current time is not the best solution. For example, instead of simply discarding a new schedule, a request for rescheduling with better estimated time, or implementing a procedure that adjusts the parameters in the function that estimates the scheduling overhead would be a better choice.

Second, a time check is made at dispatch time to make sure that each task starts according to its scheduled start time. If this check fails an alarm is raised and the alarm condition is handled by the scheduler. This alarm catches the situation where the dispatcher takes too long between stop timer and the start timer—the *dispatch out* overhead. All other timing errors at the application processor are caught by the above discussed time overrun alarm.

Finally, the timing error caused by hardware failure of an application processor is implemented. To check for hardware failures, the scheduler examines all tasks above the cutoff line (the tasks it cannot reschedule) to make sure they are done by comparing current time and scheduled finish times. If any error is detected a processor is declared faulty. Detection of this error can improve the overall robustness of the system by implementing the recovery procedure that would compensate for not executed tasks by allowing their rescheduling.

Implementation Status

All the above features are implemented except application specific error recovery routines. Our assumption is that each system would treat this error differently. We recognized a problem and provided a hook, the user should provide an error handling procedure. Note that to support some recovery methods, a scheduler code should be changed (e.g. recovery that removes some tasks and then requests rescheduling). However, in Spring, a scheduler is implemented as a user level task facilitating easy modifications and replacements.

0.2 Error Handling on the System Level

Instead of just two simple P and V routines that handle system semaphores, the *Original Spring* had a number of P and V routines. The problem was that at some unpredictable point the system would either halt or the bus error would occur. In *New Spring*, we corrected this problem, extended the error handling routine, and suggested several further improvements.

Read-Modify-Cycle (RMC) error recovery is implemented to prevent fatal failure earlier encountered when dealing with semaphores through indivisible instructions, such as Test-and-Set or Compare-and-Swap. During these instructions it is normal to detect a collision on the bus and to generate a bus error exception. To resolve this, the bus error handler has to check STAT2 (a status word on MVME136A Motorola board) for RMC error. If detected, it should set the retry bit in SSW (system status word) found on the stack, causing a hardware retry when RTE (return from exception) is executed at the end of the bus error handler.

It would be recommended to create a generic exception handler to provide better error servicing and more clues as to what went wrong. For example, this routine could look at the exception type and vector offset in the exception stack frame, and according to this information it could take the appropriate recovery action.

0.3 New Schedule Announcements

In *Original Spring*, the announcement of a new schedule could pass unnoticed by the dispatchers on application processors. Specifically, while a dispatcher is waiting for current time to reach schedule start time of the task next to execute, an announcement of a new schedule would not be noticed.

In *New Spring*, while waiting for current time to reach schedule start time of the next task, the dispatcher continuously checks a new schedule flag. If this flag is set, a dispatcher will reevaluate its waiting time according to a new schedule.

0.4 Distributed System Clock Synchronization

When a synchronized clock is needed across the nodes in SCRAMNet ring, one node, called master node, is responsible to update a value of the software clock (the clock used for scheduling) on the pre-specified location on SCRAMNet board. Due to predictable update of all SCRAMNet boards, it is possible to compensate for update delays, if necessary. This facility is required if multiple spring nodes need to share information that involves absolute system time (assignment of absolute deadlines, logging event times, both external and local activities, etc.).

In the CARTS Demo, the distributed system clock will be needed by the detection process that timestamps arrival time of detected objects, and for creation of deadlines based on consuming process needs.

0.5 Deadlines and Delays

The usual approach of specifying the deadline of the overall activity and all delays between precedence related tasks is not a sufficient way to describe the timing characteristics of some very common situations. For example, let us look into an assembly process that requires a glue job among other activities. It is possible to apply glue at any time within the given timing constraints. However, once the glue is applied, two parts must be joined before the glue dries out. That is once the chemical process is activated, the timing constraints of that chemical process must be taken into account when creating a schedule.

A simple task group deadline in assembling this product would not be sufficient. The two tasks, "apply glue" and "join parts," must be described with their local timing constraints: (1) the drying time for glue is t_1 time units, and (2) maximum time the glue can dry is t_2 time units. These timing constraints introduce not only the separation time between two activities, but also the deadline by which the second activity must complete.

This new type of timing constraints is implemented in *New Spring* in terms of *minimum and maximum delays* assigned to links between two precedence related tasks.

To fully support minimum and maximum delays, the SDL (System Description Language) is extended to handle user specification of these timing constraints. This information is later used in a scheduler to adjust release times (using minimum delay) and deadlines (using maximum delay).

0.6 Resource Handling: Request/Release Concept

In *Original Spring* the resources are required by tasks, used during task execution, and released upon task completion. This simplified schema works for some applications but it is not sufficient to model some quite common problems.

For example, in CARTS Demo the flapper can not be granted access to the linear platform between scan and pick process. Without the ability to request and release in disjoint tasks, the scan process and the pick process would have to be merged into a big task in order to keep the linear platform resource reserved. This solution is obviously undesirable, especially in more complex systems with high degree of parallelism.

Currently in *New Spring*, we are implementing the concept of request resource (RQ), release resource (RL), and request/release resource (RR). Specifically, the update eat mechanism should set the eat to infinity for a RQ, to schedule finish time (sft) for RL, and to sft for RR. If the RQ task is gone, the eat is updated to infinity if and only if we find a RL task below the cutoff line; otherwise, the eat is set to zero (meaning the RQ task is above and already gone). Their is always a precedence relationship between RQ and RL pairs, a constraints that can be easily checked by Process Activation Process (PAP module).

0.7 Resource Reclaiming

In contrast to *Original Spring*, *New Spring* employs a very rudimentary resource reclaiming schema. The scheduler continuously schedules new tasks and reschedules old tasks whenever a task arrival or completion is noticed. In this way, the resources of early finishing tasks are reclaimed as soon as they get released. The continuous scheduling/rescheduling is not considered to be a penalty, because only those tasks that are below the cutoff line are rescheduled, anyway, and the predictable execution is not disturbed. The potential disadvantage is a situation when a new event arrives while the scheduler is busy rescheduling. The probability that a new arrival will not be serviced immediately is quite high, but on the other hand, the worst case response to a new arrival is the same as if continuous scheduling/rescheduling were disabled.

To increase productivity and functionality of the scheduler, a global *done flag* is implemented. This flag is set by the dispatchers upon task completion, and reset by the scheduler when a new schedule is announced. The action of the scheduler is determined by the done flag. If the flag is set a rescheduling takes place to reclaim all resources that are freed by completed task(s). If the flag is not set, and there are no new arrivals, the scheduler is used to gather statistics and serve the monitoring interface.

0.8 RpC: Remote Process Call

In contrast to the original assumption, the CARTS Demo requires predictable and timely executions of not only hard real-time tasks, but also of vision, AI tasks, and other planners needed for robot navigation. Computation of exact WCETs of such tasks is not always possible. Quite often, these tasks require the use of some problem specific methods, languages, and specialized hardware. It is not easy to bound the loops and impose other programming constraints on these tasks just to be able to calculate WCET. The example of hard to bound real-time AI tasks are any-time algorithms. The results of these algorithms are available at any time but the quality of the result improves over time. What is the WCET of such an algorithm is hard to say.

Original Spring did not allow execution of any other tasks but spring tasks—tasks written in spring-c language. This constraint is overcome in *New Spring* by allowing spring tasks to make a remote process call (RpC). The RpC is based on interrupt capabilities of the SCRAMNet board. The Open RpC command writes into an appropriate interrupt transmit (Tx) location on the SCRAMNet board, and by doing this causes an interrupt to be transmitted. Any board in the SCRAMNet ring that has the same location set as receive an interrupt (Rx) location will receive this interrupt (see Figure 0.1). The interrupt service routine services an interrupt according to the semantics specified in the RpC table entry. Through this mechanism an execution of the external process is synchronized using the schedule created on the spring node that invoked RpC.

The spring task group shown in Figure 0.2 illustrates the specification and execution of Open RpC and Close RpC. The spring task that issues

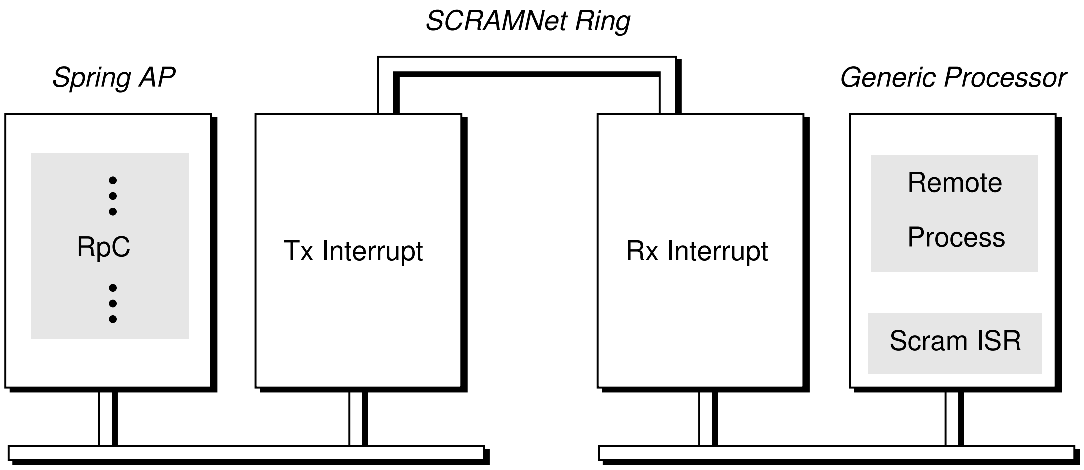

Figure 0.1: RpC Across the Nodes on the SCRAMNet Ring.

Open RpC command causes an interrupt that tells the remote process to stop waiting and start execution. The delay that follows Open RpC corresponds to the WCET of the remote process. This delay is the minimum delay between the Open RpC and Close RpC tasks. Close RpC checks the status of the remote procedure. This status is updated by the remote process and located in the corresponding RpC table entry. If this flag is set to done, the close RpC task terminates normally. Otherwise, the Close RpC intentionally executes longer than its preset WCET causing a timing error. This is a current solution that could be easily refined for future implementations to accommodate more elaborate recovery procedures for timing violations of remote processes.

Implementation

To prevent the invocation of the same process by two or more different spring tasks, and to schedule predictable execution of the remote process, each remote process is treated as a resource that is used only in the exclusive mode. The Open RpC reserves the resource with the name of the remote process, and the Close RpC releases it upon its completion.

Specification, Bootstrap, and Use

Each remote process is described in detail at the *boot master* node which

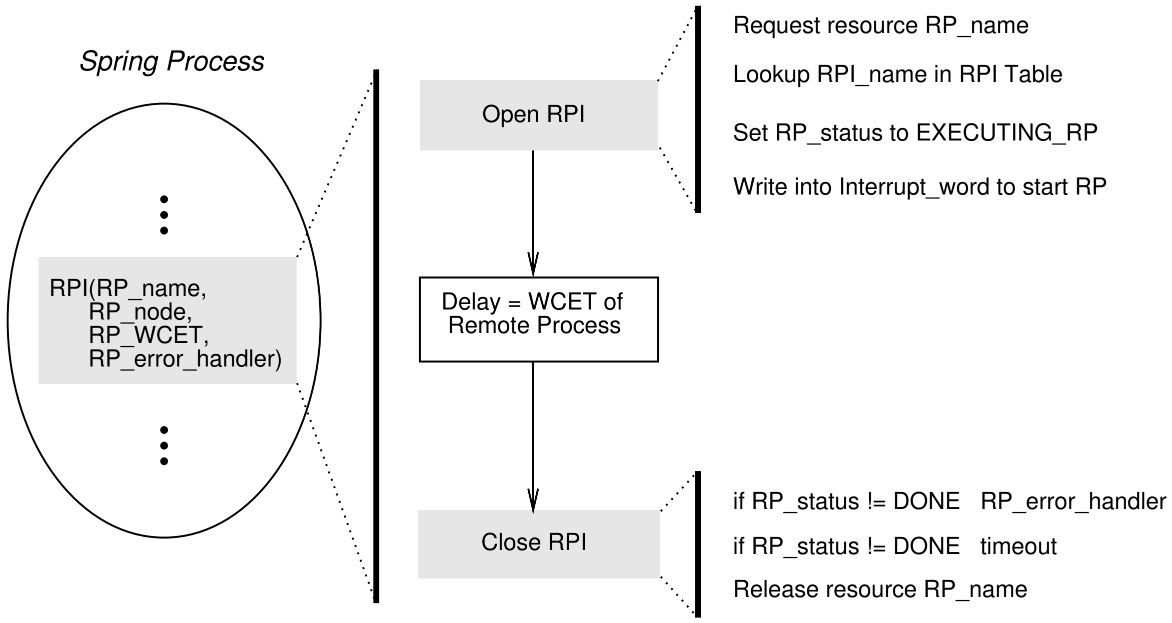

Figure 0.2: RpC within a Spring Task.

is responsible for RpC Table setup (see Figure 0.3). This description includes the specification of the remote process name, SCRAMNet setup (ACR setup) and ID of the node that issues an RpC, and ACR setup and ID of the node that executes remote process.

Once the boot node sets the RpC table, the boot process on all other nodes can commence. This is controlled by the RpC table CSR. Each node that has a remote process locates its corresponding RpC entry, reads the ACR setup, sets the interrupt location status according to ACR setup, and fills in the attributes specific to the remote process. These attributes specify how the interrupt service routine will service the interrupts generated at this RpC entry.

For example, our immediate use of the RpC mechanism has each remote process busy waiting on a local flag. The RpC table entry contains the address of the local flag and the semantics value that indicates to ISR to set this flag to let the remote process to commence its execution.

0.9 Error Handling

Close RPI calls user written error handling routine when RP_status is not set to DONE. The error handling routine is an application dependent routine. This rouitne does not prevent invocation of another remote process and it is not more constrained than any other spring routine. Actually, the invocation of another remote process is expected to be a common case when a recovery procedure is about to take place. For example, if a remote process reports in its RP_status filed that it is not done and that it is ready to freeze its data, stop current execution and start another run, the error handling routine can invoke a remote process on the same node that would tell this process to do so.

I'm not done, but if you give me one more minute I will be. This request is currently supported within OpenSpring. The way to serve such a request depends not only on the ability to perform rescheduing but also on the way the remote process is programmed. For example, if a remote process can freeze its current comoputation and get ready for a completely new run or for signal to continue the interrupted run, it is possible to handle timely executions of such tasks without violating previously created schedules.

When a schedule is created, any process, including all remote processes

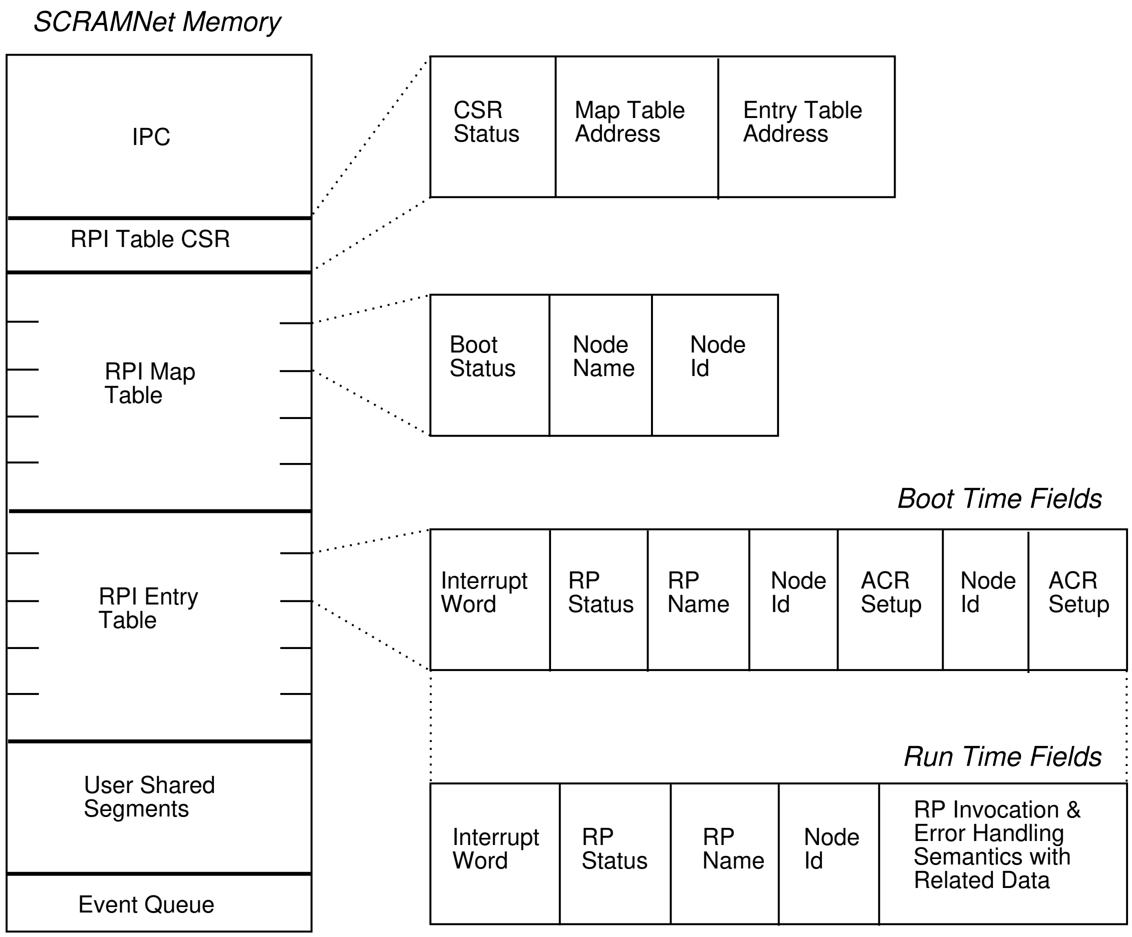

Figure 0.3: RpC Data Structure.

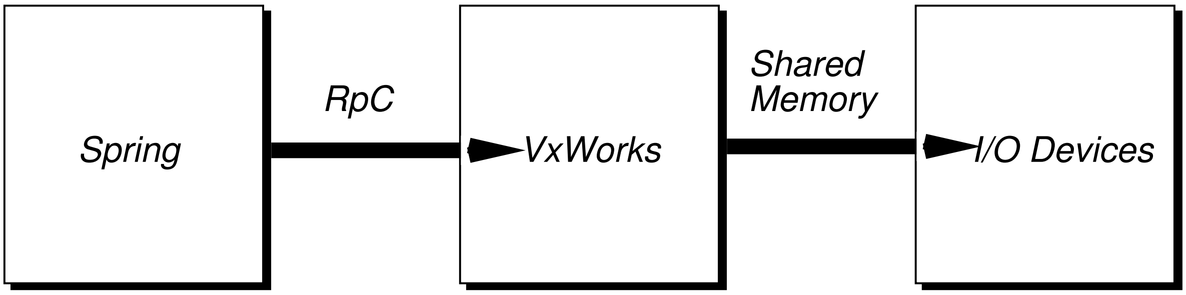

Figure 0.4: Command Chain in CARTS Demo.

as well, can be scheduled to execute back to back, that is, the second run will immediately follow the first run. We can detect that the first run will need more time either at the scheduled finish time of the first run or some time during its execution. If the need for more time is detected at the schedule finish time, the scheduler would not be able to create a new schedule without interrupting the execution sequence of previously created schedule. That is, the scheduled second run must be postponed, and due to its tight dealdine might even never execute. To prevent this, we propose the following scheme.

The invocation of the remote process consists of three major parts: Open RPI, Delay in length of the WCET of the remote process, and Close RPI. Close RPI is a procedure scheduled to run after the remote process is presumably done. If the Close RPI detects that the remote process is not done and that the user has provided an error handler routine, a call to the provided error handler routine would be made. Otherwise, Close RPI routine will time out indicating that a fatal timing violation, i.e., non-recoverable timing error has occured. User written error handler can be designed to communicate with a faulty process and to try to recover from the detected timing error. If the error is recovered, the RP_status is changed to DONE and the previously created schedule can continue execution uninterrupted. To recover from error, the error handler routine must leave the remote process in ready to start next execution state.

A faulty remote process can be prepared for next execution in more than one way. That is, aborting a process is just one and the most straight forward

CARTS Demo: Robotics Tasks

arm_controller	cage 1 board 1	I/O w/ sensors and actuators
hand_controller	cage 1 board 2	I/O w/ sensors and actuators
arm_planner	cage 1 board 3	input (x,y,z)
hand_planner	cage 1 board 4	input (close, open)
scan	* cage 1 board 5	input (color, cell) output (x,y)
flapper_controller	cage 2 board 1	I/O w/ sensors and actuators
rotating_platform_controller	cage 2 board 1	I/O w/ sensors and actuators
linear_platform_controller	cage 2 board 1	I/O w/ sensors and actuators
admittance_planner	cage 2 board 2	input: flap now rotate (speed) move (x)
detect_object	* cage 2 board 3	output (color, angle)

* VxWorks or Sparc

Figure 0.5: Robotics Tasks that Run on MVME147 of Sparc boards.

CARTS Demo: Spring Tasks

	Request	*Release*	*Request/Release*
activate linear platform			* linear platform
activate flapper 1			* flapper 1 * cell x
activate scanner	* linear platform		* scanner
activate grasp and go to home 2	* arm	linear platform	* cell y
activate deliver object and go to home 1		arm	
actvate discard object			* flapper 2

* Exclusive use of resource

Figure 0.6: Spring Tasks and their Resource Needs.

way of getting it ready for the next run. Other more complex and probably non-traditional way is to freeze the results of the current execution, estimate the time to completion, terminate the current run, and get ready for the next invocation. Using this approach, back to back runs are feasible, and what's even better, a scheduler can be consulted to try to provide resources for continuation of the interrupted run if resources are available and the constraints are not violated. It is not a straight forward to implement such a process that can continue where it stopped, but when a strict time table has to be obeyed, this is certainly a viable solution.

Another way of handling a detected timing error is to simply ignore the current data and to copy-in last set of usable data. In this case, a remote process will indicate in its status word that an old data can be used, that a current run can be aborted and that the system can continue to operate without any interruptions. This does not prevent some more elaborate techniques that register how often old data has been used, which processes should be run to

This back-to-back scheduing does not provide any time to prolong the execution of the first run.