Communicating Process Architectures 2012
P.H. Welch et al. (Eds.)
Open Channel Publishing Ltd., 2012

Supporting Timed CSP Operators in CSP++

W.B. GARDNER [1] and Yuriy SOLOVYOV
School of Computer Science, University of Guelph, Canada

Abstract. CSP++ is an open-source code synthesis tool consisting of a translator for a subset of CSPm and a C++ run-time framework. Version 5.0 now supports Timed CSP operators—timed interrupt, timed timeout, and timed prefix—as well as untimed variants of interrupt and timeout, with only 1% additional execution and memory overhead, though using interrupts is more costly. We describe the implementation and performance of the new operators, illustrating their use with a robot-vacuum cleaner case study. The tool thus becomes more useful for specifying the behaviour of soft real-time systems, and generating a timing-enabled executable program from its formal model.

Keywords. CSPm, Timed CSP, C++, code generation, software synthesis, selective formalism.

Introduction

The CSP++ project [1] is part of a vision aimed at making it easier to infiltrate a dose of formal methods into industrial practice without demanding wholesale conversion to full formal development. Its key concept, dubbed "selective formalism," allows the formal specification to be targeted at the control backbone of a concurrent system, while relegating heavy computation, data processing, and input/output to user-coded functions (UCFs) that can be written by an ordinary programmer (not a formal methods guru). UCFs are invoked at run time by linking them to events in the specification, which may result in the transfer of channel data to or from the UCF. CSP++ uses CSPm [2] to input the formal specification, and C++ for the UCFs.

CSP++ consists of two components: a translator called **cspt**, which generates C++ source code from a synthesizable subset of CSPm [3], and a set of C++ classes that provide a run-time framework with CSP semantics. Taken together, these components make CSPm specifications executable, and avoid the potentially onerous and error-prone requirement of translating a specification to compilable source code by hand.

The recommended design flow makes use of verification tools from Formal Systems Europe Ltd. [4]—checker, ProBE, and FDR2—which also process CSPm input. That input language does not provide any natural means for expressing notions of time, thus, in the "traditional" untimed CSP language, the correctness of systems is treated in terms of the order of events that programs can perform.

However, certain systems may require time as an integral component, that is, the system view will be incomplete unless it is specified within the realm of time. Numerous

[1] Associate Professor, Modeling & Design Automation Group, School of Computer Science, University of Guelph, ON, Canada, N1G 2W1. E-mail: wgardner@socs.uoguelph.ca.

examples that may require formal specification within the time domain can be found in such fields as networking, operating systems, control systems, financial transactions, etc. Since CSP++ intends to be useful for embedded systems design, it was considered essential to incorporate timing in a way consistent with CSP, as opposed to an awkward "add on" feature.

In this paper, we start by looking at some candidate methods for adding timing to CSP, and explaining why we settled on Timed CSP. Section 2 describes the requirements for the five new operators' behaviour and their method of implementation in CSP++. Section 3 uses a small case study to illustrate the new operators' use. Its CSPm specification is reproduced in an appendix, along with samples of the generated C++ source code. A major concern was that the supporting mechanisms for the new operators might unduly degrade the run-time performance of CSP++, and Section 4 reports on this. Section 5 looks at other CSP programming frameworks that support timing. Finally, we identify some future work and conclude.

1. CSP and Time

Over the years, various proposals have been made for incorporating timing into CSP. Helpful historical sources are found in [5] and [6]. A timing model was first introduced into CSP by Reed and Roscoe in their paper "A Timed Model for Communicating Sequential Processes" [7], with time being represented by non-negative real numbers. Their principle addition was the operator WAIT t ($t \geq 0$). WAIT t successfully terminates after t units of time. In a later article [8] they argued that no additional operators besides WAIT t are necessary to reflect real-time behavior of systems, including timeouts and interrupts.

Schneider essentially followed and built upon the ideas introduced by Reed and Roscoe. In [9], he gives a detailed description of the CSP language, including **Timed CSP**, which is used in this work. The Timed Computational Model introduces three operators into CSP: timeout, delay, and interrupt.

The above models assume a continuous representation of time, however, it is possible to take an alternate approach based on a periodic timing event, called **tock**, that serves as a sort of timing pulse like the ticking of a clock. One can take two views of tock [9]: (1) it represents a time evolution and takes one unit of time to happen, or (2) it is like an instantaneous drum beat, happening every unit of time.

The first interpretation of tock is not realistic for CSP++ since it is based on user-level threads, provided by GNU Portable Threads (Pth) [10], without true simultaneity. Furthermore, UCFs, or any other events, are not truly instantaneous and must take time to occur. The second tock model is more suitable, but raises the possibility of UCFs taking more than one tock (one time unit) to execute. This would delay the next tock synchronization between processes. Moreover, tock-CSP specifications are difficult to compose hierarchically, and may grow extremely long with large numbers of tocks requiring to be coded, which leads to tedious, error-prone work and hard-to-understand specifications. Even if technical and aesthetic challenges were overlooked, use of tock only directly solved one problem, representation of timed delays. Such constructs as timeouts and interrupts would still be desirable.

In the end, implementing Timed CSP operators seemed like the best choice for introducing time to CSP++. The following section describes each operator along with its implementation.

2. Timed CSP Operators in CSP++

This section covers the machine-readable notation used to express the operators, then their desired behaviour and present implementation, and also looks at one means of formal verification.

2.1 Operator Notation

Of the five new (to CSP++) operators, two were already recognized by Formal Systems tools: untimed timeout (**`[>`**) and untimed interrupt (**`/\`**). The cspt translator could simply use the same notation.

The other three required some invented notation. We decided to insert the time *t* (an integer literal) into each operator so that it would be easy to remove via regular expression processing in case one desired to launder the specification for acceptance by Formal Systems tools. Thus, we have timed prefix (**`-`*`t`*`->`**), timed timeout (**`[`*`t`*`>`**) and timed interrupt (**`/`*`t`*`\`**).

The number *t* refers to units of time, which CSP++ treats as seconds by default. The user has two ways to change the units for a particular specification:

1. Specification file statement (*u* is one of ms, sec, min, hr):
   `pragma cspt timeunit(`*`u`*`)`
2. Program execution option (one of): **`-ms -sec -min -hr`**

FDR2 accepts the `pragma` keyword, but will pass over this instance due to not recognizing `cspt`, thus the pragma can be inserted into a CSPm specification file without triggering a syntax error. Both methods can be used together. For example, the pragma can be used for translation-time specification, say minutes, then at run time, the time unit can be changed to seconds to accelerate the execution for simulation and testing purposes.

2.2 Operator Behaviour

Readers unfamiliar with the CSP meaning of timeout (**`LEFT[>RIGHT`**) and interrupt (**`LEFT/\RIGHT`**) can differentiate them by thinking like this:

- Timeout applies to whether a process is able to *start*, i.e., execute its first event, before the timeout occurs. In practice, this will only be in doubt if the first event requires synchronization with another process that has not yet offered to perform it.
- Interrupt applies to whether a process is able to *finish*, i.e., execute its last event, before the interrupt occurs.

In both cases, an occurring timeout or interrupt seizes control from the left-hand process and continues as the right-hand process. In particular, one must not think about interrupt in the conventional operating system sense, where control returns transparently to a program after the interrupt has been serviced and dismissed. In CSP usage, control cannot go back after an interrupt. Also in both cases, if the left-hand process "succeeds" (gets started, or gets finished, respectively), the right-hand process cannot get control.

The timeout or interrupt criterion can be specified either in terms of an elapsed time (for the "timed" operator) or an event (for the "untimed" operator). Thus, "timeout" is more

naturally timed, and "interrupt" is naturally untimed, though the other variants have their uses.

2.2.1 Timed Prefix

Timed prefix, also known as delay, is a special case of the prefix operator, which not only specifies the sequence in which events should be performed, but also the amount of time that must pass after one event, before the next event can be engaged in. For example,

```
S = a -3-> b -> SKIP
```

This simple specification states that process S performs event a, then waits for 3 units of time before performing event b and subsequently terminating. In practice, this means that at least 3 units of time will pass between events a and b.

The generated C++ code is quite straightforward. It simply calls a "nap" function that blocks for 3 units of time. (Because the nap needs to be interruptible, as discussed below in Section 2.2.4, the underlying implementation does a timed wait on a condition variable.) Due to the nature of Pth scheduling, it is understood that this will result in a *minimum* delay before the process receives control of the CPU again, and not a hard timing constraint.

2.2.2 Timed Timeout

Timed timeout can be considered a case of "choice" similar to deterministic choice. For example,

```
S = a->P [3> RIGHT
```

The first event a of the left-hand side is offered. If it succeeds, then the choice is resolved, the timeout will not apply, and the process continues as P. If event a blocks awaiting synchronization, a 3-units timer is started, then whichever event occurs first—the synchronization or the timer—resolves the choice. In case of timeout, the pending synchronization on event a is cancelled and the offer to perform that event is withdrawn, then S continues as RIGHT.

Note that, just as with deterministic choice, CSP++ requires the first event of the left-hand process to be exposed, e.g., `a->P [3> RIGHT`, not just `LEFT [3> RIGHT`. This is to help the translator identify a single event that must be offered, which is easy enough for the programmer to code, but can potentially be tricky to discover at translation time if LEFT has a complex definition.

2.2.3 Untimed Timeout

Untimed timeout is, perhaps surprisingly, a nondeterministic operator by nature. A simple example is found in [9] where the offer of a cheap sales price can lapse at any time before the buyer takes it up. Consider `a->P [> b->Q`. This system can evolve in two ways: (1) it can remain in the state where both events a and b are on offer, which is equivalent to deterministic choice, `a->P [] b->Q`; or (2) it can "timeout" into the state where b occurs and the system continues as Q.

From a *descriptive* standpoint, which evolution, (1) or (2), takes place is unpredictable and effectively subject to an internal decision. From a *prescriptive* standpoint more compatible with our software synthesis objective, such nondeterminism is unhelpful. Therefore, we decided to give this operator a consistent deterministic treatment. The code

generated always tries event a first. If a succeeds immediately (i.e., without blocking for synchronization), the operator functions as (1) deterministic choice and continues as P. In that scenario, b is not offered at all. But if a would block, it chooses the timeout route (2) and the offer of a is withdrawn. Then when b occurs, the process will continue as Q.

This interpretation has the ability to produce all traces of **`a->P [> b->Q`**, while providing additional useful software functionality beyond that of the already-available deterministic choice operator (**`[]`**). To be clear, with **`a->P [] b->Q`**, the system is committed to performing whichever event a or b occurs first. But with **`a->P [> b->Q`**, if event a is not ready, then the choice is resolved in favour of **`b->Q`**. Thus, untimed timeout offers a kind of "polling" operator, consistent with the notion of somehow "timing out" when the initial attempt does not succeed. As with the timed timeout, we require only the first event of the left-hand side to be exposed.

2.2.4 Untimed Interrupt

In contrast to timeout, where a choice will decide which process to execute, interrupt implies concurrent execution. Consider this simple specification of untimed interrupt,

```
S = P /\ e -> Q
```

Process P starts its execution and tries to run to completion. If process P does complete, event e and subsequently process Q will never run. However, if at any point during P's execution event e happens, any further progress of P must be cancelled, and the system must continue with process Q. Conversely, process P must finish its execution prior to event e in order to avoid cancellation.

Interrupt operators proved to be the hardest to implement, partly due to the fact that CSP++ processes were never designed to be interrupted, based, as they are, on non-preemptive GNU Pth threads. Non-preemption means that once a particular thread of execution gets control of the CPU, it will only relinquish it if a blocking situation occurs, such as input/output or waiting for an event, or the thread may explicitly yield control in a cooperative fashion. In brief, the two-part solution was (1) to make every blocking method in the framework interruptible, i.e., capable of unblocking before the condition it originally blocked on is fulfilled; and (2) to enable the unblocked method to abort its process by means of a C++ exception. More details will be given below.

Consider the basic interrupt pattern **`LEFT /\ inter->RIGHT`**. (Note that the interrupting event has to be exposed for the translator.) By convention we start execution with the right-hand side of interrupt operator, i.e., test to see if the interrupting event can happen right away. In that case, LEFT should be prevented from starting.

If inter does not succeed and wants to block, that will relinquish control of the CPU and give LEFT a chance to run. But then, given the non-preemptive GNU Pth scheduler, how can the interrupting event inter happen while process LEFT is running? For an interrupt to take place, LEFT must block at some points during its execution, otherwise inter will never get a chance to run.

Let us clarify what it means for process LEFT to be interrupted: LEFT may spawn a large subprocess tree. For LEFT to be interrupted means that no further events can execute anywhere in the subprocess tree, i.e., appear in the system's trace, after the interrupt. Every process in LEFT's subtree must terminate and join with any threads that are waiting for them. Eventually, process LEFT will join with the threads it spawned and will also terminate, thus effecting the interrupt.

In practice, to implement `S = P /\ e->Q`, a minimum of two threads are needed: one thread to run process P, and another thread to attempt to execute the interrupting event e. In CSP++, process S would already have its own thread of execution. Therefore, we let S try event e, and, if necessary, spawn a new thread for process P. This solution uses the minimum number of threads and avoids unnecessary overhead.

The initial design for the interrupt operator was focused around Pth thread cancellation techniques, that is, having the thread that executes the interrupting event cancel the thread(s) executing P. Unfortunately, the complexity of insuring proper cleanup of P and its subprocesses—especially withdrawing from attempted synchronization—became intractable. A better solution shifted the focus from the interrupting process to the interrupted one to do its own cleanup and then self-terminate. The latter required performing a "short circuit" return to the function-caller that executes each process definition. The C++ exception mechanism worked well for this purpose, taking care of unwinding the stack and thereby invoking any associated object destructors, which automatically free the dynamic storage under control of stack variables defined in the generated code.

2.2.5 Timed Interrupt

Implementation of the timed interrupt, `LEFT /t\ RIGHT`, follows the same logic as the untimed interrupt with one exception: an interrupting event is no longer needed, as the interrupt occurs due to time elapsing. As a special case, if the interrupt time was specified with t=0, then we chain to the interrupting process right away, never even considering the left-hand process.

2.2.6 Summary of CSP++ Restrictions

Compared to Timed CSP, the timeout operators in CSP++ force the user to expose the first event of the left-hand side process. This matches our longstanding treatment of the deterministic choice operator.

Compared to CSP, the untimed interrupt operator in CSP++ forces the user to expose the first event of the right-hand process. This means that the interrupting process cannot progress alongside the left-hand process, performing (non-interrupting) internal events, but rather begins its execution once the exposed interrupting event happens.

A key restriction stemming from the use of a non-preemptible threading library (GNU Pth) is that an interrupting event or elapsed timer may not have an effect of the left-hand side process immediately. Once that process gets control of the CPU, it will only release it if a blocking situation occurs. The case study in Section 0 illustrates specifying delays to allow for process interruptibility.

2.3 Verification

Formal verification and state space exploration tools such as FDR2 (as of version 2.91) and ProBE cannot directly be used to verify the correctness of a timed system as they do not support Timed CSP. The system could be stripped of any timing information, of course, and then FDR2 and ProBE can be applied. However, sometimes the added timing information can significantly change a system's behaviour, and the formally verified untimed version of the same program will not guarantee the absence of deadlocks, livelocks, or other race hazards in the timed counterpart. Some other means is needed.

A team at the National University of Singapore School of Computing has been developing HORAE [11], a tool used to reason about Timed CSP specifications using

Constraint Logic Programming (CLP). We could not obtain the tool itself, but we can show, based on its published description, how one can input timed specifications using the new CSP++ operators when it becomes available.

HORAE, shown in Figure 1, encompasses operational and denotational semantics of Timed CSP encoded in CLP(R), the constraint solver, as separate modules. The two modules are used to reason about different properties of Timed CSP. The denotational module (denoeng) captures the timed traces and timed failures of CSP and is used to check the timewise refinement properties. The operational module (opereng) captures the "evolution relations and timed event transition relations of a process" [11] and is used to verify variable bound properties. Both modules are responsible for checking the safety and liveness of a given Timed CSP specification.

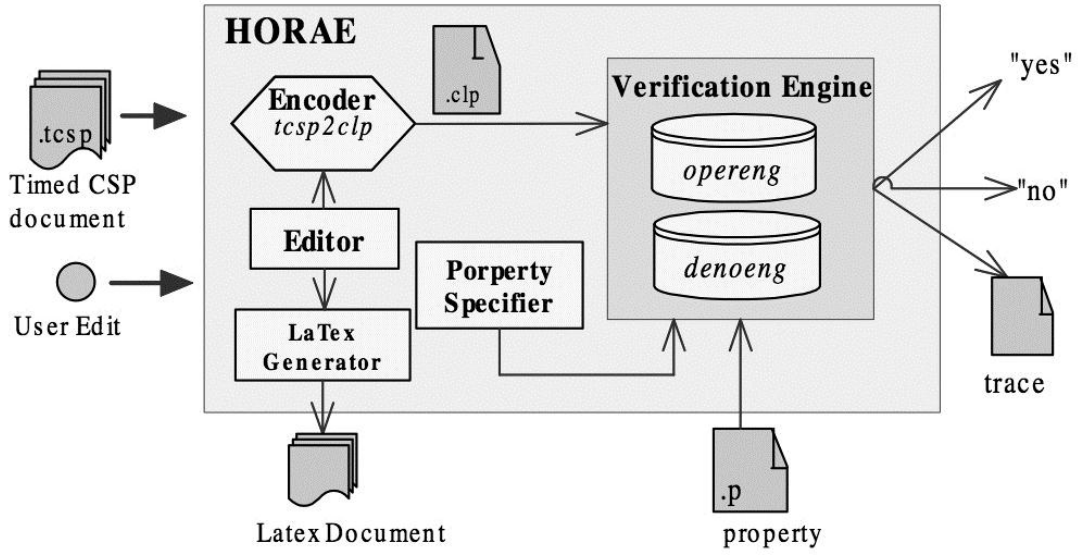

Figure 1. HORAE Design Flow [11]

Table 1 presents HORAE syntax for Timed CSP operators. The symbol t represents an integer constant holding a time interval value.

Table 1. CSP++ and HORAE Syntax for Timed CSP Operators

Operator	CSP++ Syntax	HORAE .tcsp Syntax
Delay	`a -t-> b`	`a ->{t} b`
Untimed Timeout	`P [> Q`	*not supported*
Timed Timeout	`P [t> Q`	`P |\{t} Q`
Untimed Interrupt	`P /\ a->Q`	`int(P, a, Q)`
Timed Interrupt	`P /t\ Q`	`tint(P, t, Q)`

HORAE syntax for interrupt operators resembles function calls, while we tried to make CSP++ syntax look like CSPm's as closely as possible, especially to maintain compatibility with FDR2 for the untimed operators it does recognize. Even though differences in syntax between HORAE and CSP++ are evident, it is fairly easy to make specifications written for CSP++ work with HORAE. A simple script can accomplish just that by searching specifications written for CSP++ for syntax incompatible with HORAE and restructuring it accordingly. Development of such a script was left for future work when HORAE is finalized and released for general use.

We were able to obtain Timed CSP specification files that had been used to test the HORAE tool. These specifications described small, popular case studies: Dining

Philosophers, Timed Vending Machine, and Timed Railroad Crossing. After adjusting the syntax to fit CSP++ we ran these specifications through our system to get executable versions. These experiments showed that HORAE and CSP++ are very compatible. Full specifications of the three Timed CSP case studies are presented in [12].

3. Vacuum cleaner case study

A case study called VAC was created to demonstrate the newly-supported operators. VAC simulates a simple robot-vacuum cleaner, which, if left in automatic mode, scoots around a room picking up dust and avoiding obstacles, but can also be operated manually through a remote control. Following the pattern of earlier case studies [12], the VAC design consists of two parts, or design models—a functional model and an environmental model. The VAC specification consists of processes describing different physical parts of the system that belong to the functional model. Environmental entities providing stimuli to VAC, such as a user and room layout, belong to the environmental model. The latter is very useful during system design and testing stages, when providing physical input to the system might be too expensive or the right environment may simply be unavailable. When the simulated system is ready to be implemented, the environmental model can be removed from the design, and processes and channels described in the functional model can interact directly with the physical environment by means of UCFs. The philosophy of employing these multiple models in CSP for synthesis by CSP++ is explained in detail elsewhere [1].

Figure 2 illustrates VAC's interaction with the environment. Time units are in seconds. The VAC design includes one functional model, the VAC itself, and two environmental models—a simulated room with dust in which VAC operates, and a user providing extra input.

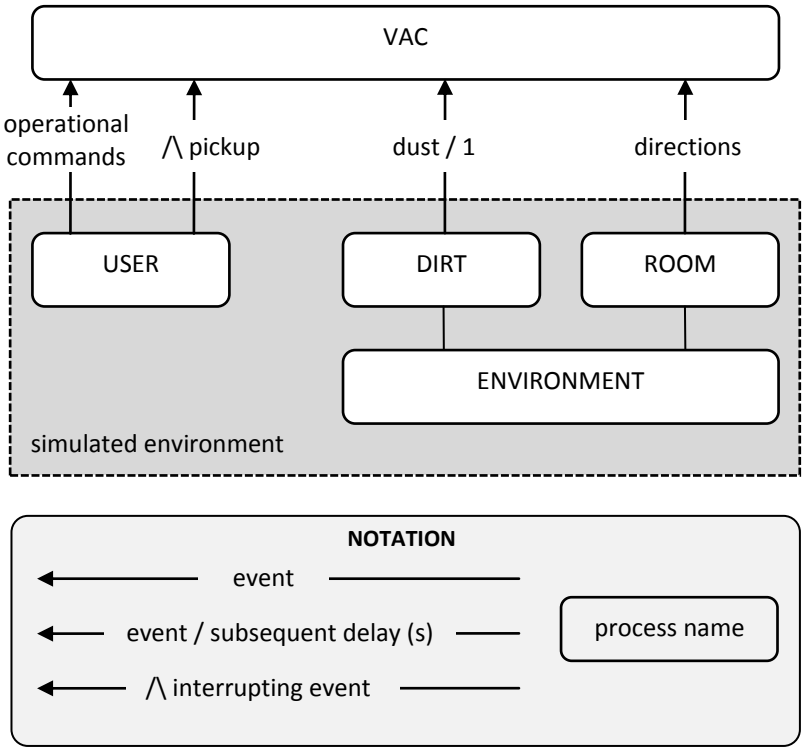

Figure 2. VAC Interaction with the Environment

The VAC case study was executed with a number of variations in the environment to exercise different execution paths. The appendix shows the CSP specification for VAC (channel declarations omitted) interspersed with translations of selected statements illustrating all the new operators. The complete specification including several user-coded functions are found in [12] Appendix D.

In creating the functional model, instead of focusing on complex robot intelligence that actually does detect and avoid obstacles while figuring out its path through the environment, we tried to show how to use Timed CSP operators in combination, and how these primitives contribute to the overall behaviour of the specified system. First, let us consider Figure 3 which visually portrays the overall design of VAC using StateCharts. Note that the numbers appearing in the notation in place of "d" and "subsequent delay" all signify time intervals in seconds. This visual representation of VAC gathers all interacting components in one place and helps grasp the final plan before it is written in CSPm.

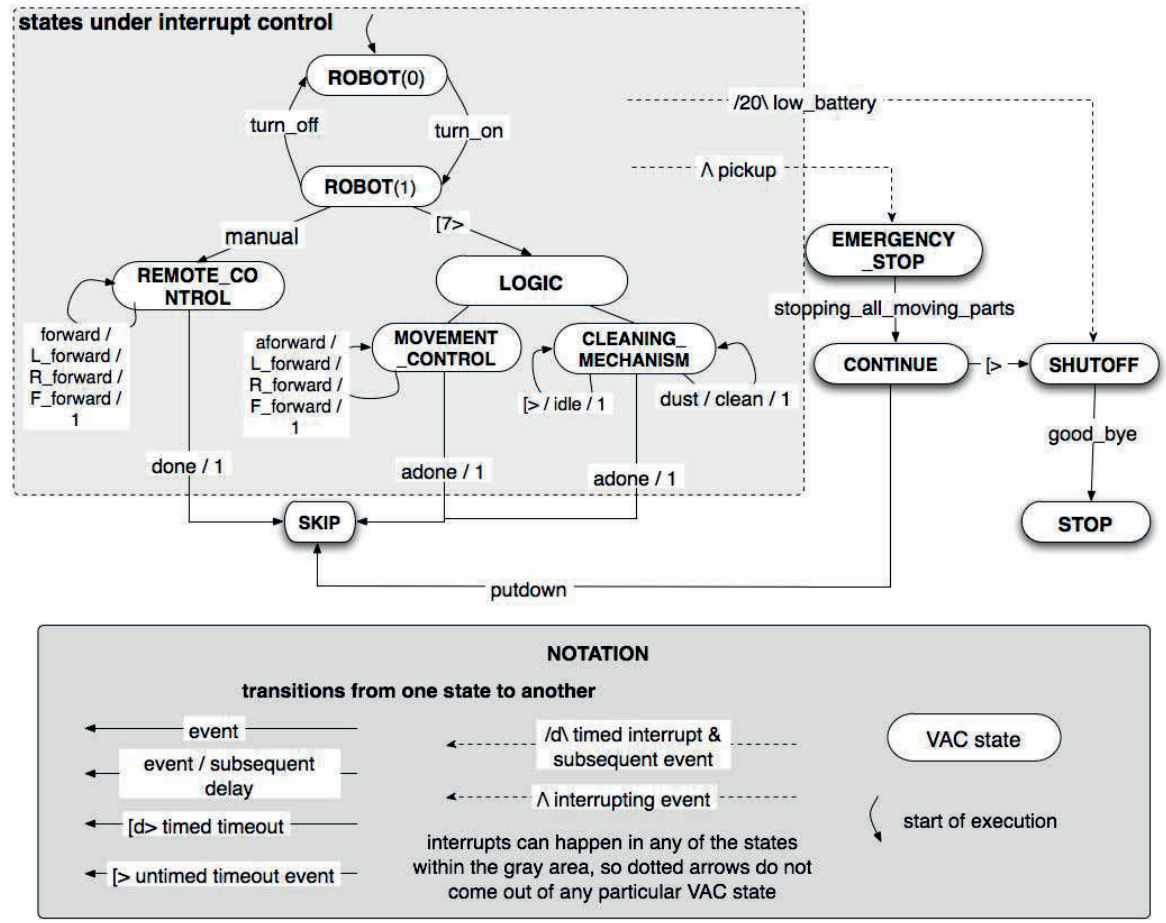

Figure 3. VAC StateChart

The use of each of the new operators within VAC will now be highlighted in the following subsections. The statements and generated code can be read in context by consulting the appendix.

3.1 Timed prefix

Timed prefix is a special case of the prefix operator, which not only specifies the sequence in which events should be performed, but also the amount of time that must pass between finishing of one event and subsequent attempt to execute the next.

MOVEMENT_CONTROL and REMOTE_CONTROL processes illustrate the use of timed prefix. In MOVEMENT_CONTROL, after the choice is made and appropriate wheel actions are triggered, each recursive call to MOVEMENT_CONTROL or successful termination is delayed by one time unit. The delays give interrupts a chance to execute.

The REMOTE_CONTROL process, which synchronizes with the user commands, models VAC movement as directed by the user. REMOTE_CONTROL also features one-unit time delays that are included for the same reason as in MOVEMENT_CONTROL.

3.2 Timeouts

VAC illustrates the use of timeout operators as follows:

```
WHICHOPMODE = (manual -> REMOTE_CONTROL) [>
  ((turn_off -> ROBOT(0))  [7> AUTOMATIC_MODE)
```

After the user turns the VAC on, the system goes into the state where the subsequent mode of operation will be decided. If the user wants to operate the robot manually then the manual command will trigger the robot's operation via REMOTE_CONTROL. However, if no command follows, WHICHOPMODE will timeout giving the user other options. Essentially, the polling operation is performed on the manual event to see if the user wanted to do some manual operations, and if not, the system moves on to another choice situation under timed timeout. At this point the user has the option of turning the robot off, or else, if no command is received within seven time units, the robot will operate in AUTOMATIC_MODE.

The process describing the cleaning mechanism of VAC features two nested untimed timeouts performing sequential polling operations:

```
CLEANING_MECHANISM = (adone -1-> SKIP) [>
  ((dust -> clean -1-> CLEANING_MECHANISM) [>
        (idle -1-> CLEANING_MECHANISM))
```

At first, the process checks if the adone command is received, triggering successful termination. If not, the robot checks for dust on the floor, cleans it and goes back to the original state. If neither of the first two events occurred, the robot remains idle for one time unit and goes back to the original state ready to perform the timeout checks again.

Untimed timeout adds additional expressive power to CSP++. Deterministic choice (`[]`) has implicit priority as it tests its operands from left to right, executing the first event that can successfully be completed. If CLEANING_MECHANISM were rewritten using deterministic choice, CSP++ would attempt to execute adone, dust, and then idle events to see which one succeeds first. If none of the three events was successfully executed during the first attempt, the thread will block until one of the events participating in deterministic choice succeeds. That is, the process would remain committed to all three events. But with the timeout operator, we only have to try to execute the left-hand side event to see if it succeeds right away. If not, we "timeout" to the right-hand side with no additional attempts to execute the left hand side event. This "polling" operation adds a new design feature to CSP++, not available before.

3.3 Interrupts

Untimed interrupt is illustrated in the following example:

```
RUNNING = WHICHOPMODE /\ pickup -> EMERGENCY_STOP
```

At any point in time, while the robot is performing its cleaning routines, the user may pick the robot up. This presents a safety-critical situation as the robot has moving parts which may injure a person. Hence, stopping the working robot upon pickup is ideal for using the untimed interrupt. In the above specification, event pickup interrupts the running process WHICHOPMODE and its subprocesses, chaining to EMERGENCY_STOP which simulates the actual stopping of all moving parts.

The VAC specification also demonstrates the use of the timed interrupt operator to simulate the robot's battery life. All batteries last only so long, so we can use the timed interrupt to specify how long the robot may perform its cleaning routines:

```
ROBOT(1) = RUNNING /20\ low_battery -> SHUTOFF
```

In this case, simple passage of time triggers the low battery interrupt. After the robot is turned on, it only has 20 time units to perform its duties. After 20 time units, VAC's execution will be interrupted and process SHUTOFF will get control, simulating a dead battery.

4. Performance

After the implementation of timed operators in CSP++, it was important to make performance measurements to see how the new version of the tool compared to the previous untimed version. We conjectured that implementation of the interrupt operator in particular had to add noticeable overhead to the framework, since using exceptions forces the C++ compiler to generate information needed to carry out stack unwinding, destructor invocation, and so on. In contrast, timeouts are simply another form of deterministic choice, as far as the framework is concerned, with no additional overhead, while timed prefix does not burden specifications that do not use it. Thus, it was worthwhile investigating what price all users would have to pay for support of the new operators.

The performance comparisons between untimed CSP++ (v4.2) and timed CSP++ (v5.0) were executed on a 1.8 GHz AMD Athlon(TM) XP 2500+ processor with 1.5 Gb of memory running Kubuntu 8.04 with Linux kernel v2.6.24. The g++ compiler used during the tests was gcc-4.2.3 with -O2 optimization, and GNU Pth version 2.0.7.

The compiled programs were run without the tracing flag "-t", but with the quick exit flag "-q", which avoids printing a dump when the system executes STOP.

Each test was run 21 times with the average of the last 20 being used for comparisons. The first run was discarded to account for the effects of paging. Execution times were obtained using the Linux 'time' command. The sum of the user and system times was used for comparison. The largest standard deviation for any group of 20 runs was 0.12 seconds, while the average standard deviation for all of the tests was 0.06 seconds.

All tests were run using some variation of the Disk Server Subsystem (DSS) case study developed by Gardner [14], which has been a de facto benchmark for measuring the performance of CSP++. The DSS features a parallel composition of the disk server and a number of clients sending requests to the disk server and receiving acknowledgements. The code for all tests is given in [12].

4.1 Untimed vs. Timed CSP++

To compare performance of the two versions of CSP++, variations of the DSS case study were used as tests. Note that DSS does not utilize the new timed operators, so this comparison was intended to reveal any general increased burden on execution time as a result of utilizing C++ exceptions in v5.0.

Each test execution involves 5000 iterations, which each recreate the interleaved client processes, therefore the number of thread creations is proportional to the total requests. This is a severe measure of framework overhead, as very little that can be called "useful work" is done.

The three test cases are: (1) 2 clients, total of 10,000 requests; (2) 4 clients, 20,000 requests; (3) 8 clients, 40,000 requests.

Table 2. Performance of CSP++ v5.0 vs. v4.2

	Test 1	**Test 2**	**Test 3**
v5.0	11.77 s	33.92 s	117.84 s
v4.2	11.73 s	33.49 s	116.93 s

The results show that CSP++ v4.2 is slightly faster (0.3-1.3%). Our hypothesis was that the difference in execution time would be due to using the C++ exception handling mechanism compiled into v5.0 of CSP++. However, the GNU gcc compiler manual states that "GCC will generate frame unwind information for all functions, which can produce significant data size overhead, although it does not affect execution." Further investigation demonstrated that run times for CSPm specifications compiled with and without the -fno-exceptions flag produced no appreciable difference. Therefore, the difference seen in Table 2 can be attributed mainly to additional actions performed in the v5.0 framework. These include checking whether a new CSPm process is within the scope of an interrupt operator, and checking for interrupted status when any blocked thread awakens. This marginal additional overhead in the framework is borne by all specifications whether they use interrupts or not.

4.2 Interrupts in Specifications

It was also useful to investigate how much overhead inclusion of the interrupt operator adds to a given specification. For this experiment, the DSS case study included two client processes performing a total of 10,000 disk requests. Test case 1 is simply the original DSS without any interrupt operator. Test case 2 places an interrupt at the top level with sufficient time so that it will not occur before the test finishes. That is, the interrupt operator is just "there" without being activated.

Table 3. CSP++ v5.0 Impact of Interrupt Operator

	No `/t\`	**With `/t\`**
v5.0	11.77 s	12.51 s

As can be seen, the mere inclusion of the interrupt operator increases execution time by 0.74 seconds or 6.3%. In this scenario, both interleaved processes—which are each recreated 5000 times—will find themselves within the scope of the top-level interrupt operator, meaning they will have to add themselves to the interrupt environment object's waiters' list, and more flag checking will be triggered during event processing.

4.3 Memory Cost of Using C++ Exceptions

As shown in Section 4.1, enabling C++ exceptions only increased execution overhead around 1%. Nevertheless, the gcc manual stated that the use of exceptions may "significantly" increase data size. To see exactly how much memory overhead was produced, we generated the same DSS test using three compiled versions of the CSP++ framework: (1) v4.2 compiled with -fno-exceptions; (2) v4.2 with –fexceptions; (3) v5.0 with –fexceptions. (-fexceptions is the default for gcc 4.)

Table 4. Executable size with and without exceptions enabled

	v4.2	**v4.2**	**v5.0**
Exceptions?	**No**	**Yes**	**Yes**
size (KB)	258.7	261.5	288.3

The increase in size due to enabling exceptions in version v4.2 is 2.8 KB, which amounts to a mere 1.1%. When we look at the results of v4.2 and v5.0, the difference is greater and can be attributed to the code added to support the new operators.

In conclusion, contrary to our expectations, the additional execution and memory overhead attributable to supporting the new operators is minimal.

5. Related work

CSP was never intended to be a programming language. Therefore, it may not be surprising that code generators based on CSP input are relatively few. One active project utilizes the NOCC compiler to generate code designed to execute with the KRoC run-time system [15]. Its input language, MCSP, is a different, smaller subset of CSP than CSP++ recognizes, but its strength is the ability to handle millions of concurrent processes. It achieves this with special attention to efficient process memory allocation, in contrast with CSP++ which relies on Pth for thread management and does not presently have the replication syntax to express creating a quantity of some process.

A number of other projects have focused on implementing the CSP formalism in a popular programming language. Here we will compare CSP++ with projects developed at the Computing Laboratory of the University of Kent, namely, C++CSP2 [16] and JCSP [17]. Unlike CSP++, the two libraries do not provide automatic code generation from a CSP specification, however, they implement a range of CSP constructs aimed at easing concurrent programming. The libraries are analogous to CSP++'s object-oriented back-end framework which is based on a class library implementing CSP components, especially process and channel. The goal of this study, presented in full in [12], was to compare and contrast how a programmer may use CSP++, JCSP and C++CSP2. Here we report only results related to timing constructs.

Some timing abilities are easy to obtain in the JCSP library. For example, the timed prefix operator is not mentioned explicitly in the API documentation, however, its effect appears in several examples. The programmer may instantiate a CSTimer object, and then execute its sleep(t) method to produce a delay of t milliseconds. This method has the same semantics as java.lang.Thread.sleep.

Another example is timed timeout, which CSP++ treats as a kind of choice. Therefore, it makes sense to find that JCSP can produce this effect using an AltingBarrier, CSTimer.setAlarm(), and Alternative.priSelect() (to discover whether the synchronization barrier or timeout was triggered).

JCSP does not provide interrupt facilities which correspond directly to CSP operators. However, it is possible to use Alternative and CSTimer to come up with interesting scenarios that do resemble interrupting behaviour.

One example reads in data from an array of input channels for a programmer-defined amount of time. Once the time elapses, the process abandons reading and terminates. This resembles Timed CSP's timed interrupt to some extent. However, it is not a true interrupt, because the loop that keeps calling Alternative.fairSelect() has to also keep manually checking whether the timer event has occurred. By contrast, in CSP++ the checking is done internally by the framework, and the looping process will be terminated automatically.

C++CSP2 treats timing constructs in the same way as JCSP.

Another library in a similar vein is CTC++ [18] from the University of Twente. As with the Kent libraries, programmers code directly in the target language and may instantiate CSP components such as Process and Channel<T> (for a specific data type T). CTC++ supports higher-level features beyond "plain" CSP, e.g., buffered channels, exceptions, prioritized parallel, barriers, and much more. A number of timing utility functions are provided. There is no translator from CSP specifications per se, but the tool gCSP [19] can be used to construct a graphical process/channel system model, which can be generated in CSPm for verification using FDR2, in occam, or in C++ for compilation with CTC++ and then execution.

6. Future work and Conclusion

With the addition of operators from Timed CSP, CSP++ is now suitable for soft real-time systems synthesis. When a timed operator is encountered in a given specification, CSP++ can guarantee that, at a minimum, the specified amount of time will pass before the next action will be taken, and that at least the specified time will apply to timeouts and interrupts.

Future work will concentrate attention on UCFs. They do not have any special mechanism for blocking, though they are free to utilize Pth-wrapped system calls that block only the calling thread rather than the entire operating system process. This means that UCFs are not well-integrated with interrupts at present. A UCF-linked event is currently not interruptible, and a UCF-linked event that blocks is not suitable to be an interrupting event, because, unlike for non-UCF events, there is no "try/retry" mechanism for UCFs. This is a comment on the general limitations of CSP++'s interface to UCFs, which warrants further study.

As for hard real-time systems—those whose correctness or even safety depends on responding to events within tiny latencies (e.g., milliseconds or less)—the main stumbling block for using CSP++ is the lack of timing constraints in Timed CSP. There are no constructs to specify that, for instance, event b *must* meet some deadline, say within n time units of a previous event, or that process P must execute with a specified frequency. Even if there were, the current design of the CSP++ framework leaves the underlying threads package to carry out non-preemptive scheduling according to its own algorithm. Since CSP++, even enhanced with Timed CSP operators, cannot guarantee timing constraints, it does not aspire to be a tool for synthesizing hard real-time systems.

Timed CSP operators are available in CSP++ since v5.0, which is available for free download from the project's website.[2] The cspt translator and the run-time framework are licensed under GPL (GNU General Public License) and LGPL (GNU Lesser General Public License), respectively. As a bonus, a Python script, fdrscript.py, collects the trace

[2] CSP++ home page, http://www.uoguelph.ca/~gardnerw/csp++/.

output generated by a CSP++ program (run with the "-t" option), massages it into a form accepted by FDR2, and appends the statement 'assert SYS [T= TRACE' which can then be run through FDR2 to verify that the actual system trace is valid for the original CSP specification.

Acknowledgments

This research was supported by NSERC (Natural Science and Engineering Research Council) of Canada.

References

[1] William B. Gardner. Converging CSP specifications and C++ programming via selective formalism. ACM Transactions on Embedded Computing Systems, 4(2):302–330, 2005.
[2] FDR2 User Manual. Formal Systems (Europe) Ltd, and Oxford University, 2010.
[3] W.B. Gardner. CSP++: How Faithful to CSPm? Communicating Process Architectures 2005 (WoTUG-28), Eindhoven, Sep. 18-21, Concurrent Systems Engineering Series, pages 129-146. IOS Press.
[4] Formal Systems (Europe) Ltd. Website [online, cited 6/1/12]. Available from: http://www.fsel.com/.
[5] Jim Davies and Steve Schneider. A brief history of Timed CSP. In MFPS '92: Selected papers of the meeting on Mathematical foundations of programming semantics, pages 243–271, Amsterdam, The Netherlands, 1995. Elsevier Science Publishers B. V.
[6] Joël Ouaknine and Steve Schneider. Timed CSP: A retrospective. This paper is electronically published in Electronic Notes in Theoretical Computer Science, 2005 [online, cited 6/1/12]. Available from: http://www.computing.surrey.ac.uk/personal/st/S.Schneider/papers/entcs06.pdf.
[7] G. M. Reed and A. W. Roscoe. A timed model for Communicating Sequential Processes. Theor. Comput. Sci., 58(1-3):249–261, 1988.
[8] G. M. Reed and A. W. Roscoe. The timed failures-stability model for CSP. Theor. Comput. Sci., 211(1-2):85–127, 1999.
[9] Steve Schneider. Concurrent and Real-time Systems: The CSP Approach. John Wiley & Sons, Ltd., 2000.
[10] GNU Portable Threads (Pth), 2006 [online, cited 7/20/12], Available from: http://www.gnu.org/software/pth/
[11] J. S. Dong, P. Hao, J. Sun, and X. Zhang. A reasoning method for Timed CSP based on constraint solving. In 8th International Conference on Formal Engineering Methods (ICFEM'06), Macau, November 2006.
[12] Yuriy Solovyov, Extending CSP++ Framework with Timed CSP Operators, masters thesis, Dept. of Computing and Information Science, University of Guelph, 2008.
[13] Stephen Doxsee. Reengineering CSP++ to conform with CSPm verification tools. Master's thesis, University of Guelph, 2005.
[14] W.B. Gardner. CSP++: An Object-Oriented Application Framework for Software Synthesis from CSP Specifications. PhD thesis, Department of Computer Science, University of Victoria, Canada, 2000.
[15] Frederick Barnes, Compiling CSP. Communicating Process Architectures 2006 (WoTUG-29), Edinburgh, Sep. 17-20, Concurrent Systems Engineering Series, pages 377-388. IOS Press.
[16] Neil Brown. C++CSP2: A many-to-many threading model for multicore architectures. Communicating Process Architectures 2007, pages 183–205, 2007.
[17] Peter Welch, Neil Brown, James Moores, Kevin Chalmers, and Bernhard Sputh. Integrating and extending JCSP. In Alistair A. McEwan, Steve Schneider, Wilson Ifill, and Peter Welch, editors, Communicating Process Architectures 2007. IOS Press, 2007.
[18] B. Orlic and J.F. Broenink. Redesign of the C++ Communicating Threads library for embedded control systems. In F. Karelse, editor, 5th Progress Symposium on Embedded Systems, pages 141-156, Nieuwegein, NL, 2004. STW Technology Foundation.
[19] D.S. Jovanović, B. Orlic, G.K. Liet, and J.F. Broenink. gCSP: a graphical tool for designing CSP systems. In I. East, Jeremy Martin, P.H. Welch, David Duce, and Mark Green, editors, Communicating Process Architectures 2004 (WoTUG-27), volume 62, pages 233–252, Amsterdam, September 2004. IOS Press.

Open Channel Publishing Ltd., 2012

Appendix

This is the CSPm specification for VAC. Generated C++ source code for selected statements is shown in boxes directly below the corresponding CSPm. The code at line 27 illustrates timed prefix and untimed timeout, at line 63 timed interrupt, at line 86 untimed interrupt, and at line 109 untimed and timed timeout. For simulation purposes, the environmental model consists of movements induced by the ROOM, occurrences of DIRT, and USER interventions, which all have timing inserted via timed prefix. The system execution trace will be printed by running with the "-t" option. User-coded functions linked to these events are not shown here (see [14]).

```
1  MOVEMENT_CONTROL = (aforward -> L_forward -> R_forward -> F_forward -1->
2  MOVEMENT_CONTROL) []
3              (abackward -> L_backward -> R_backward -> F_backward -1->
4  MOVEMENT_CONTROL) []
5              (aleft -> L_backward -> R_forward -> F_turn -1->
6  MOVEMENT_CONTROL) []
7              (aright -> L_forward -> R_backward -> F_turn  -1->
8  MOVEMENT_CONTROL) []
9              (astop -> L_stop -> R_stop -> F_stop -> MOVEMENT_CONTROL)
10       []
11             (adone -1-> SKIP)
12
13 REMOTE_CONTROL =
14 (forward -> L_forward -> R_forward -> F_forward -1-> REMOTE_CONTROL)
15                          []
16 (backward -> L_backward -> R_backward -> F_backward -1-> REMOTE_CONTROL)
17                          []
18 (left   -> L_backward -> R_forward -> F_turn  -1-> REMOTE_CONTROL)
19                          []
20 (right  -> L_forward -> R_backward -> F_turn  -1-> REMOTE_CONTROL)
21                          []
22 (done  -1-> SKIP)
23
24 CLEANING_MECHANISM = (adone -1-> SKIP) [>
25 ((dust -> clean -1-> CLEANING_MECHANISM) [>
26 (idle -1-> CLEANING_MECHANISM))
```

```
27 AGENTPROC( CLEANING_MECHANISM_ )
28    Agent::startDChoice( 1 );
29       adone();
30    if ( Agent::whichUTChoice() == 0 ) {
31       Agent::nap( 1*timeunit );
32    }
33    else {
34       Agent::startDChoice( 1 );
35          dust();
36       if ( Agent::whichUTChoice() == 0 ) {
37          clean();
38          Agent::nap( 1*timeunit );
39          CHAIN0( CLEANING_MECHANISM_ );
40       }
41       else {
42          idle();
43          Agent::nap( 1*timeunit );
44          CHAIN0( CLEANING_MECHANISM_ );
45       }
```

```
   }
   END_AGENT;
}

AUTOMATIC_MODE  = ENVIRONMENT
[|{|aforward, abackward, aleft, aright, adone, astop, dust|}|]
                  LOGIC

LOGIC = MOVEMENT_CONTROL [|{|adone|}|] CLEANING_MECHANISM

EMERGENCY_STOP = stopping_all_moving_parts -> CONTINUE
CONTINUE = (putdown -> SKIP) [> SHUTOFF
SHUTOFF = good_bye -> STOP

ROBOT(0) = turn_on -> ROBOT(1)

ROBOT(1) = RUNNING /20\ low_battery -> SHUTOFF
```

```
AGENTPROC( ROBOT_c1 )
   int time_=20*timeunit;
   if( !time_ ) {
      low_battery();
      CHAIN0( SHUTOFF_ );
   }
   else {
      timed_int_r.interrupt();
      CHANGEPRIO( 5 );
      Agent* a5 = START0( RUNNING_, 0 );
      int choice_ = Agent::startTI( a5, time_ );
      CHANGEPRIO( 0 );
      WAIT( a5 );
      Agent::popEnv( 1 );
      if( choice_ == 1 ) {
         low_battery();
         CHAIN0( SHUTOFF_ );
      }
   }
   END_AGENT;
}
```

```
RUNNING = WHICHOPMODE /\ pickup -> EMERGENCY_STOP
```

```
AGENTPROC( RUNNING_ )
   Agent::startDChoice( 1 );
      pickup();
   if ( Agent::whichUIChoice() == 0 ) {
      CHAIN0( EMERGENCY_STOP_ );
   }
   else {
      pickup_r.interrupt();
      CHANGEPRIO( 5 );
      Agent* a6 = START0( WHICHOPMODE_, 0 );
      int choice_ = Agent::startUI( a6 );
      CHANGEPRIO( 0 );
      WAIT( a6 );
      Agent::popEnv( 1 );
      if( choice_ == 1 ) {
         CHAIN0( EMERGENCY_STOP_ );
      }
   }
   END_AGENT;
}
```

```
WHICHOPMODE = (manual -> REMOTE_CONTROL) [> ((turn_off -> ROBOT(0)) [7>
AUTOMATIC_MODE)
```

```
AGENTPROC( WHICHOPMODE_ )
   Agent::startDChoice( 1 );
      manual();
   if ( Agent::whichUTChoice() == 0 ) {
      CHAIN0( REMOTE_CONTROL_ );
   }
   else {
      Agent::startDChoice( 1 );
         turn_off();
      if ( Agent::whichTTChoice( 7*timeunit ) == 0 ) {
         CHAIN1( ROBOT_c0, 0 );
      }
      else {
         CHAIN0( AUTOMATIC_MODE_ );
      }
   }
}
```

```
-------------------ENVIRONMENTAL MODEL--------------------------
ENVIRONMENT = ROOM ||| DIRT

ROOM = aforward -1-> aleft -1-> aforward -1-> aright -1-> abackward -1->
adone -> SKIP

DIRT = dust -1-> dust -2-> SKIP

USER = turn_on -10-> pickup -> putdown -> SKIP

SYS = ROBOT(0)
[|{|turn_on, turn_off, manual, autom, forward, backward, left, right,
done, pickup, putdown|}|]
      USER
```