EXTENDING CSP++ FRAMEWORK WITH TIMED CSP OPERATORS

A Thesis

Presented to

The Faculty of Graduate Studies

of

The University of Guelph

by

YURIY SOLOVYOV

In partial fulfilment of requirements

for the degree of

Master of Science

April, 2008

© Yuriy Solovyov, 2008

ABSTRACT

EXTENDING THE CSP++ FRAMEWORK WITH TIMED CSP OPERATORS

Yuriy Solovyov
University of Guelph, 2008

Advisor:
Professor W. B. Gardner

Communicating Sequential Processes (CSP) is a formal process algebra used to specify and reason about concurrent systems. Timed CSP was subsequently created to add the capabilities of delays, timeouts, and interrupts, necessary for specifying systems with time-sensitive properties. The tool CSP++ was developed to make machine-readable CSPm specifications directly executable via automatic C++ code generation, and extensible via user-coded functions. Formerly, CSP++ could only synthesize CSP specifications abstracted from any timing information. In this work, CSP++ was extended to translate and synthesize specifications with new operators drawn from Timed CSP. These new features are demonstrated with a VAC Automated Cleaner case study and compared to other CSP-inspired programming libraries.

Acknowledgements

I would like to express much gratitude to people who made this research possible:

- First and foremost my parents and my family for their unconditional love and belief in my success.
- My supervisor, Dr. William Gardner, for his excellent guidance and kindness in and out of school.
- My friends for their encouragement and support.

Table of Contents

List of Figures

List of Tables

1 Introduction

The goal of this research was to extend the capabilities of the existing tool, CSP++ [Gar05], to synthesize C++ code from a specification including Timed CSP [Sch00] operators. This provides system designers using CSP++ with the new ability to specify delays, timeouts, and interrupts. In this introduction, we first give background on CSP++ and the benefits of automatically generating source code from a formal specification. Next, we define the problem of incorporating Timed CSP and the immediate motivations for tackling it. Finally, the research contributions of this thesis are summarized.

1.1 Background

It is hard to underestimate the role software plays in our everyday lives. Sadly, there are examples of situations when poorly-developed software was directly responsible for significant damages to property and infrastructure or even loss of human life. Examples of such events include the Therac-25 radiation therapy machine [LT93] which due to software race conditions claimed the lives of five people, and the North American Blackout of 2003 [Pou04]. To avoid such extreme situations or simply to make software more reliable, developers may turn to formal methods which allow them to reason about software and expose unwanted behaviour such as deadlocks, livelocks, race conditions and other resource starvation hazards when no meaningful progress is made by the system.

Commercial tools exist which allow formally verified systems to be checked and reasoned about. Unfortunately, formal specifications, even after being subjected to verification tools, cannot guarantee the absence of any revealed flaws in the final software product. There are several reasons for that. First, the life cycle of many software products, especially those running on expensive and unique hardware, reaches years and often decades. During the life cycle, software undergoes numerous changes and upgrades which may lead to inconsistencies with the original formally-verified design. Second, there are no guarantees that hand-translation of formal specifications to the final software implementation did not introduce any errors, or the programming language of choice supported all the features used in the formal specifications.

Automated software synthesis tools can be used to battle these drawbacks. Given that the automated translation process is correct, such code synthesis tools can provide guarantees that the formal specifications and the final implemented products will stay in-sync. Any changes made to the formal specification will automatically make it into the final product.

Development of such a tool, based on Communicating Sequential Processes (CSP), has been the primary research focus of W.Gardner and his students at the University of Guelph. CSP++ [Gar00, Gar03] was originally created as part of W.B.Gardner's Ph.D. thesis work at the University of Victoria. It was subsequently updated and extended by Stephen Doxsee, as part of his Master's thesis [Dox05], and by Joshua Moore-Oliva. What sets CSP++ aside from other software synthesis tools or CSP-inspired libraries is a combination of things: the ability to make formal specifications written in CSP directly executable in the form of generated C++ code (hence the tool's name), and extensibility of the generated code with User-Coded Functions (UCFs). The tool provides the ability to formally specify and subsequently generate C++ code for only the critical parts (responsible for process synchronization and interaction) of software products being designed. Furthermore, parts of the system responsible for simple Input/Output operations or any calculations can be plugged into the formal specification via user-coded functions saving developers from needless efforts to specify those parts formally.

Until recently, CSP++ was unable to synthesize code from Timed CSP specifications, i.e., specifications that include the notion of time, where not only the order of performed operations is important, but also the times at which those operations take place. The rest of this chapter explains the motivation and the approach taken to reengineer the CSP++ tool to include Timed CSP operators. The roadmap for the rest of the thesis is also presented.

1.2 Problem Definition and Motivation

Although CSP++ features a wide range of CSP operators and constructs and is quite capable of synthesizing and running C++ code derived from formal CSP specifications, it could not do so for CSP specifications which included the notion of time. Numerous examples that may require formal specification within the time domain can be found in such fields as networking, operating systems, financial transactions, etc.

In the “traditional” untimed CSP language, the correctness of systems is treated only in terms of the order of events which programs can perform. In reality, however, it is hard to completely abstract from the notion of time. Certain systems may require time as their integral component, that is, the system view will be incomplete unless it is viewed within the realm of time.

With regards to safety-related specifications, it may be necessary to analyze systems considering their real-time behavior. Unfortunately, the traditional CSP language cannot accommodate such demands. Thus, it was necessary to expand CSP with a new semantic model, which would include timing constructs. Using such a model allows the analysis of the correctness of CSP processes with regard to time, i.e., correctness of a system would shift from the simple order in which it performed events to include concrete times at which it performed these events [Sch00].

In order for CSP++ to become a more attractive tool in the software development field, it had to be extended with new features allowing the tool to be used in a wider range of applications. Considering the fact that CSP++ already features most of the CSP constructs, aside from non-deterministic operators, the decision was made to fully explore the previously unexploited area of Timed CSP.

We say *fully* explore Timed CSP because until this research, CSP++'s ability to deal with timed formal specifications was very limited. The untimed CSP language uses a special *tock* event to mark the passage of time, and it represents one abstract time unit. It can be thought of as an internal clock regulating the passage of time. To represent a time delay between execution of two successive events, a number of *tock*s would have to be inserted between them. Such treatment of time in the traditional CSP language can lead to very confusing and hard-to-read formal specifications of timed systems, and is completely unacceptable for specification of large and complex systems.

Timing operators, specifically designed to represent time, were a perfect fit and a great extension of the untimed CSP language. Timed CSP made even complex formal specifications easier to read and understand. Furthermore, when implemented in CSP++, new timed operators, such as delay, untimed and timed timeouts, and untimed and timed interrupts, would allow users to have greater control over their formal specifications and the generated code.

Dr. Michael Alexander, a professor at Vienna University of Economics and Business Administration, Austria, whose research focuses on using formal methods to model transactions between financial agents, expressed interest in using CSP++ as the tool for generating an executable back-bone in his formal models [Ale05]. This further strengthened the decision to research the possibility of implementing timed CSP operators in CSP++ making it a more appealing and versatile automated software synthesis tool. Aside from financial transactions, a whole range of soft real-time applications could adopt CSP++ once it was extended with timing operators.

1.3 Research Approach and Contributions

Research presented in this thesis had to be divided into several stages in order to achieve the desired goal of adding CSP timing operators to the existing CSP++ tool.

Much time and thought was dedicated to researching the history of the Timed CSP language, as there is no unified standard. As CSP was never intended to be a programming language per se, it was necessary to decide what CSP paradigms it would actually be possible to implement in CSP++. Our implementation of Timed CSP tries to follow Schneider's interpretation [Sch00] as closely as possible, however, some paradigms were left unimplemented. Detailed discussion of what Timed CSP features made it into CSP++ and why is presented in Chapter 3. We also present reasons why some features were left unimplemented.

CSP++ can be divided into two main parts: a translator, called **cspt**, which parses a formal CSP specification and creates corresponding C++ objects, and the back-end Object-Oriented Application Framework (OOAF), which actually runs the translated code. Significant thought was given to what type of Timed CSP syntax our tool should support. To this point, CSP++ conformed to CSPm syntax. CSPm is a machine-readable dialect of CSP. It is supported by several commercial tools developed by Formal Systems, Ltd. [For]: Failure-Divergence Refinement 2 (FDR2), used to expose or verify absence of any deadlocks, livelocks, or other properties in a CSP specification; Probe Behaviour Explorer (ProBE), used to explore possible execution paths in a given CSP specification; and Checker used for type checking. As useful as these tools are for untimed CSP specifications, they cannot be used to reason about Timed CSP specifications as they only support two of the operators we set out to implement in this research. It should be noted that the aforementioned tools are still the primary tools used for verification of untimed CSPm specifications. However, a tool that supports the whole range of timed CSP operators was needed. Fortunately, a research team at the National University of Singapore is currently working on a Timed CSP verification tool, called HORAE [DZSH06]. Chapters 2 and 4 show how our Timed CSP syntax can be adjusted to work with HORAE.

CSP++'s back-end also underwent changes to accommodate the newly added timed operators. The greatest challenge was the implementation of interrupt operators. CSP++ uses GNU's non-preemptible Pth threading library to run synchronous processes. Due to the non-preemptive nature of the threading library it was necessary to figure out a way to interrupt, i.e., stop execution of a running thread, so that the interrupting process can take over. The answer came in the form of C++ exceptions. It should be noted that only blocking operations (operations that force a running thread to go to sleep and be woken up upon execution of a desired event or a signal) of a running thread can be interrupted. A more detailed and technical discussion of how timing operators were implemented in CSP++ will be found in Chapter 3.

After timing operators had been implemented, it was necessary to demonstrate the new capabilities of the reengineered CSP++ tool. A case study featuring VAC, an automated household vacuum, was developed. The study implements the full range of timed CSP operators and shows how they can be combined to reach the desired system behaviour. A number of much smaller regression tests were added to the system to make sure subsequent work on the OOAF or the translator will not break or alter the existing CSP++ features.

Finally, for the sake of better measuring CSP++ against “the competition,” a study has been conducted to compare CSP++ with CSP-inspired libraries.

The rest of the thesis is organized as follows: Chapter 2 provides the background on the Timed CSP language, presents a brief history of CSP++, lists verification tools used to reason about Timed CSP specifications and gives a survey of competing CSP-inspired libraries. Chapter 3 discusses considered approaches and challenges encountered during the implementation of timed operators in CSP++, and is the core of this research. In Chapter 4, the VAC case study is presented. Chapter 5 compares the performance of untimed CSP++ with the new timed version. Chapter 6 gives a more detailed comparison between CSP++ and competing CSP libraries. Chapter 7 concludes this thesis with a summary of achieved results and possible future work.

2 Background and Related Work

Information provided in Chapter 2 is meant to help the reader more easily understand the CSP language and its timed extension, Timed CSP. A brief history of CSP++ is presented to show the technical origins of this research. We also survey available CSP verification tools and competing projects aimed at bridging the CSP formalism with conventional programming languages.

To avoid any confusion, it is important to distinguish between some of the key terms used in this chapter. **CSP**, or Communicating Sequential Processes, refers to the formal method developed by Tony Hoare and first introduced in 1978 [Hoa78]. **Timed CSP** refers to the updated CSP language which includes all of the original CSP constructs as well as new timed constructs aimed at describing time-sensitive systems. **CSPm** is a machine-readable dialect of CSP supported by a number of commercial tools and CSP++. The CSP++ synthesis tool, first developed by W.Gardner, currently a professor at the University of Guelph, as part of his Ph.D. thesis work [Gar00] and extended as part of this thesis, consists of two parts, the **cspt** translator (called hereafter “the translator”), and the object-oriented application framework (“the framework”).

Chapter 2 will begin by presenting an overview of CSP in Section 2.1 to prepare the reader for better understanding of the material presented in this thesis. Sections 2.2 and 2.3 will extend the reader’s knowledge of CSP with different timing models. Section 2.4 will discuss formal verification of Timed CSP specifications. Section 2.5 will give an overview of CSP++ to set the tone for Chapter 3 and detailed implementation details. Finally, Section 2.6 will survey CSP inspired libraries, followed by a more detailed discussion in Chapter 5.

2.1 Overview of Communicating Sequential Processes

Communicating Sequential Processes was specifically designed to formally describe and reason about distributed and concurrent systems. These systems may be very complex and usually consist of many parts which execute in parallel and synchronize at certain points. Such parallel execution may lead to phenomena not present in sequential programs, in particular, deadlocks, livelocks, and race conditions. Formal specification of concurrent systems using CSP can help reveal such pitfalls before they make it to the final product.

We will use a simple vending machine example to illustrate some of the CSP constructs to make the reader more comfortable with CSP syntax and formal specifications. The operators are explained in the sections below.

$$\begin{aligned}&\text{VM} = \text{coin} \rightarrow (\text{soda} \rightarrow \text{VM} \;\square\; \text{return\_coin} \rightarrow \text{VM})\\&\text{USER} = \text{coin} \rightarrow \text{soda} \rightarrow \text{SKIP}\\&\text{USER} \parallel \text{VM}\end{aligned}$$

2.1.1 Processes

Every system described by CSP is defined in terms of processes that perform certain events. These processes may engage in communication and synchronization with one another. The vending machine example above consists of two processes: a vending machine (VM), and a customer purchasing a soda (USER). The two processes are composed in parallel.

Simple processes, such as USER, are defined by their names, '=' sign, and a series of events they can perform separated by the '→' prefix operator. More complex processes may be defined in terms of other processes.

Every process has a notion of *alphabet* associated with it. It is the set of all possible events that a process can engage in. For instance, VM's alphabet is {coin, soda, return_coin}. It contains three events, however, every iteration of VM executes only two of the three events. This brings us to the notion of traces. Every process's execution leaves a so-called *trace*, or a sequence of events it engaged in. For example, VM's trace in the vending machine specification would be <coin, soda>.

2.1.2 Process Termination, Looping and Chaining

In CSP every process definition has to end with another process. CSP defines two processes, SKIP and STOP, which do not produce any action, but rather define either successful termination (SKIP) or a deadlock situation (STOP). For instance, the Vending Machine example has a USER process, which performs actions coin and soda, and then successfully terminates with SKIP.

It is possible in CSP to define processes in terms of themselves to accommodate recursion. The VM process in the Vending Machine specification is an example of such tail recursion, where VM, after accepting a coin and dispensing a soda, returns to its original state and is ready to perform those actions again.

To define finite loops, CSP employs the concept of parameterized processes, for example:

```
COUNTDOWN(n) = tick → COUNTDOWN(n-1)
COUNTDOWN(0) = time_is_up → SKIP
COUNTDOWN(5)
```

Every iteration of COUNTDOWN performs a tick event and decrements variable n that it passes to itself. When n reaches zero, COUNTDOWN performs its last iteration and successfully terminates. The trace of COUNTDOWN(5) would be <tick, tick, tick, tick, tick, time_is_up>.

CSP also allows for specifications where upon successful termination one process can continue as another, i.e., sequential composition. For instance,

STUDENT = SCHOOL; HOMEWORK
SCHOOL = class1 → break → class2 → SKIP
HOMEWORK = reading → break → writing → SKIP

When SCHOOL process is successfully terminated with SKIP, STUDENT immediately goes into HOMEWORK mode, and executes reading, break, and writing events, and also successfully terminates.

2.1.3 Choice

In CSP one can describe systems which may have several possible execution paths. Which execution path a process takes is determined by input from its environment, and the concept is dubbed external choice. The vending machine example illustrates the use of external choice, defined by the ‘ □ ’ operator:

VM = coin → (soda → VM □ return_coin → VM)
USER = coin → soda → SKIP
USER || VM

After accepting a coin, VM is ready to either dispense a soda or return_coin before going back to its original state. Input from USER (in this case synchronization on soda) is what determines what path VM takes.

CSP also defines internal choice where no input from the environment is necessary for a system to determine which path to follow. The system resolves the choice internally. Being a nondeterministic operator, the internal choice operator ‘ $\sqcap$ ’ is not very useful in software synthesis, but does allow developers to describe a wider range of systems.

2.1.4 Event Renaming and Hiding

So far, we have only used CSP specifications in which events perform simple actions. CSP, however, supports more complex event actions, such as event renaming and event hiding.

In certain specifications it makes sense to hide particular events from the specification interface, i.e., make them internal to the process. This is very similar to the object-oriented concept of encapsulation. Below is an updated Vending Machine example, which illustrates event hiding:

```
VM = (coin → calculate_change →
        (soda → VM  □  return_coin → VM) ) \calculate_change
USER = coin → soda → SKIP
USER || VM
```

The calculate_change event in the VM process above is made internal to the process and is denoted by backslash ‘\’ followed by the set of events to be hidden. There is no need for the USER to see how VM performs calculate_change as long as the correct amount of change is returned. The event calculate_change will not appear in a trace of VM.

At times it may be desirable to reuse a certain portion of specification code, however, under different names. CSP can do just that with event renaming. The following example is adapted from [Sch00]:

ISABELLA = isabella.get_pencil → isabella.draw →
isabella.drop_pencil → SKIP
KATE = *f*(ISABELLA)
ISABELLA ; KATE

where *f*(isabella.*x*) = kate.*x* and *f*(y) = y if y is not of the form isabella.*x*

Say we want to model behaviour of two girls, ISABELLA and KATE, who get a pencil, draw a picture, and then drop the pencil. Both girls perform the exact same events only under different names. It makes sense to model only ISABELLA's behaviour and then rename all of the events to suit KATE's needs.

Line KATE = *f*(ISABELLA) of the above specification says that process KATE is equal to process ISABELLA, i.e., it has the same set of events, however, each isabella event should be renamed, by the given renaming function, to kate in process KATE. Execution of the two processes sequentially will result in the following trace <isabella.get_pencil, isabella.draw, isabella.drop_pencil, kate.get_pencil, kate.draw, kate.drop_pencil>.

2.1.5 Parallel Execution

As already mentioned, Communicating Sequential Processes was specifically developed to describe and reason about concurrent systems, whose processes execute in parallel. Two major types of parallel execution supported by CSP are *synchronized* and *interleaved*.

Synchronized composition (also called interface parallel) means that all participating processes may execute independently until an event present in all of the processes' alphabets (set of events a process can potentially engage in) is encountered. Going back to the Vending Machine example at the beginning of Section 2.1, processes VM and USER are composed in parallel with their synchronization points being soda, coin, and return_coin. This means VM and USER must perform events coin and soda simultaneously, and if one of the processes arrives at the synchronization point before the other, it must wait so that both may cross the "checkpoint" together.

Another type of parallel execution is interleaved parallel where two or more processes can execute in parallel, but completely independently. No synchronization, except for termination, is allowed for participating processes, even in situations where they contain the same events in their respective alphabets. For example,

ENGINE ||| WINDSHIELD_WIPERS

A car engine and windshield wipers have different functions and operate completely independently of one another with absolutely no need for synchronization, except for termination: when a car is turned off, both the engine and the windshield wipers stop working.

2.1.6 Channels and Process Communication

In CSP processes can not access each other's private data directly, and any communication and data transfer between processes is restricted to the use of channels, which can be regarded as a special case of process synchronization. A channel is defined as synchronous non-buffered unidirectional communication between a pair of processes. This implies that CSP processes must block until data is transferred from one end to another. Note that machine-readable CSPm dialect, and CSP++, lift the restriction for unidirectional communication between a pair of processes, meaning it is possible to have a broadcasting situation where several processes may read the same data from one channel output by a single process.

To illustrate the concept, we will enhance the vending machine example with simple non-buffered unidirectional data transfer between two processes:

```
VM = (coin?x → if (x > 1) then calculate_change → change → soda → VM
          else if (x==1) then soda → VM
          else not_enough → VM ) \calculate_change
USER = coin!2 → change → soda → SKIP
USER || VM
```

To distinguish between channel communication and simple synchronization, special characters are used to signify input/output between processes. In the example above, the USER writes 2 to channel coin signified by '!' sign, while the VM process reads ('?') the data on the same channel into variable x. Variable x can be further accessed throughout that process's execution and can possibly be passed to another process via parameterized processes.

The example above also illustrates other operators available in CSP, such as if/else selection and relational expressions.

2.2 Extension of CSP with Timing Operators

Timed CSP is a real-time extension of the traditional CSP language. While the CSP language is a great tool for describing and analyzing concurrent and parallel systems, it provides no facilities for analyzing systems with regard to their real-time behavior. Often correctness of systems may depend on their performance within the time domain; examples of such programs include networking protocols and operating systems. Consequently, timing was introduced into CSP. It was first proposed by Reed and Roscoe in 1986 [RR88]. Timed CSP language has evolved through many changes since its inception. A brief history of Timed CSP may be found in [DS95] and [OS05]. The following subsections present the main characters and events in the Timed CSP history.

2.2.1 Roots of Timed CSP: G.M. Reed and A.W. Roscoe

A timing model was first introduced into CSP by Reed and Roscoe in their paper “A Timed Model for Communicating Sequential Processes” [RR88]. Their original requirements for the timing model included:

- *Continuous with respect to time.* Time should be represented by non-negative real numbers and there should be no minimal time difference between two consecutive visible events performed by two processes executed asynchronously in parallel.
- *Realistic.* A process is only able to perform finitely many events in a finite amount of time.
- *Continuous and distributive with respect to semantic operators*. “All semantic operators should be continuous, and all the basic operators ..., except recursion, should distribute over nondeterministic choice” [RR88]. Consider an example of distribution over nondeterministic choice:

  $$a \rightarrow (P1 \sqcap P2) = (a \rightarrow P1) \sqcap (a \rightarrow P2)$$

  An observer cannot distinguish whether the internal choice was made before or after the performance of event a.
- *Verifiable design*. The model should provide means for verification of time critical systems.
- *Compatible*. The model and the proof system associated with it should simply be an extension of the already existing framework of the untimed CSP language.

The authors, in effect, added only two new operators to untimed CSP:

- $WAIT\ t$, where $t \geq 0$. $WAIT\ t$ successfully terminates after t units of time.
- $\perp$, a diverging process that does not engage in any visible event with the environment. Consider, a simple CSP process $P = a \rightarrow P$. If we wish to hide event a from the envi-

ronment, then $P\backslash a = \perp$. Essentially, $\perp$ did not add any new semantics to CSP, but allowed for easier representation of diverging processes.

Reed and Roscoe also formulated several timing postulates or basic assumptions about timing as related to the domain of a distributed system. The major assumption was the inclusion of δ, a system delay constant, to ensure the requirement for realism: only finitely many events can be performed in a bounded time. For example, process $a \rightarrow P$ initially performs event a and is then ready to engage in process P. δ introduced a short delay following the occurrence of event a, describing a requirement for the overall process to recover from execution of event a and prepare to execute P. Recursive statements followed the same idea, where every recursive call would be delayed by δ. Another postulate stated that "the order of events which happen at the same time is irrelevant" [RR88]. Both assumptions were later dropped, which led to a simpler and more coherent semantics of the timed language. In a later article, "The Timed Failures-Stability Model for CSP" [RR99], the authors reiterated the above-mentioned postulates with the exception of the delay constant. They also argued that no additional operators besides WAIT t are necessary to reflect real-time behavior of systems. If it is possible to use a combination of the original CSP operators with WAIT t to produce timeouts and interrupts, adding only one timed construct to the untimed language is, essentially, sufficient to specify time-critical systems.

It should be mentioned, however, that the current semantics of interleaved execution in CSP requires all interleaved processes to synchronize on termination. This requirement led to the impossibility of representing interrupts using traditional CSP operators. One interleaved process cannot interrupt another, as the interrupted process is required to remain alive and wait for the interrupter to synchronize on termination. The requirement for interleaved processes to synchronize on termination is very reasonable and can be illustrated by a simple example. Imagine a car: Windshield wipers and turn signals operate independently of each other, running as interleaved processes. However, when the ignition key is turned off these processes do synchronize on termination.

2.2.2 Research by Steve Schneider

Early articles by Steve Schneider essentially followed and built upon the ideas introduced by Reed and Roscoe. Thus there are many similarities. We will only highlight the main points. As mentioned above, Timed CSP introduced a non-zero delay δ imposing a minimum time interval between any two events. For example, process $Q = a \rightarrow b$ has a time delay δ between events a and b. The implicit inclusion of δ was especially valuable during declaration of recursive calls. For instance, $N = a \rightarrow N$ was time-guarded and could not perform infinitely many a events in no time. Subsequent evolution of timed CSP eliminated δ from the alphabet of timed CSP. Currently, all delays associated with process executions have to be made explicit. The two processes mentioned above are now written as $Q = a \rightarrow (WAIT\ d; b)$ and $N = a \rightarrow (WAIT\ d; N)$. More information on the original timed CSP can be found in [DS88].

In [Sch00], Steve Schneider gives a detailed description of the CSP language, including Timed CSP. The Timed Computational Model introduces three operators into CSP. They are **timeout**, **delay**, and **interrupt**. Before these constructs are analyzed in more detail it is necessary to point out important assumptions used in the timed computational model:

- *Instantaneous events*. Events are performed at an instant the process is ready to perform them and take no time to execute. Events that take time to complete may be modeled as an event at which action begins and an event at which it ends.
- *Newtonian time*. "Time passes with reference to a single conceptual global clock" [Sch00].
- *Real-time*. Time can only be represented in terms of real non-negative numbers.
- *Maximal parallelism* represents the notion of each process having its own processor that it executes on. Any scheduling analysis must be made explicit.
- *Maximal progress*. An event must occur at an instant all synchronizing parties are ready to perform it. This assumption also includes urgency of internal events. They must be performed at the instant they become available.
- *Finite variability*. Only a finite number of events can be performed in a finite amount of time.
- *Synchronous communication*. Synchronization events require simultaneous participation of all involved parties.

Possible executions of CSP processes are viewed in terms of *transitions* and *evolutions*. Event transitions do not include any notion of time and are only concerned with instantaneous execution of events. Evolutions, on the other hand, are used to describe the flow of time. Such separation allows for seamless integration of time into an existing framework of untimed CSP operators. "Delays and durations in a process description must then be given explicitly, and are not implicitly bound up with individual operators" [Sch00].

An important property that must be considered is that process evolutions are deterministic, that is, a process can reach only one state by simply allowing time to pass. Any other state reached in the same amount of time implies introduction of event transitions somewhere along the way. The passage of time alone does not introduce any new possibilities for system behavior.

The passage of time can be represented using the following evolution transition $Q \overset{d}{\rightsquigarrow} Q'$, where d can be any positive real number: $d > 0$. Such notation describes evolution of process Q to process Q' which takes d units of time. Furthermore, it is possible for a process to undergo a series of states $\{Q_i\}$ over all time without performing any events described as $Q \overset{\infty}{\rightsquigarrow}$, which can only appear as the final step in the execution. In this case Q is said to be stable[1], since it will not perform any internal events, i.e., not visible to the environment. It must be noted that $Q \overset{\infty}{\rightsquigarrow}$ is not an evolution transition as it can never be completed.

2.2.3 Timed CSP Operators

STOP

Process STOP represents a deadlock situation, at which point the system is not able to perform any internal or external events. It is only logical to conclude that passage of time has no influence on the STOP process, i.e., STOP may allow any amount to time to pass and it will still remain STOP.

SKIP

Process SKIP is used to denote immediate and successful termination of a system. The process is immediately ready to terminate, and it remains ready as time passes. $SKIP \overset{\infty}{\rightsquigarrow}$ denotes a possibility when SKIP is prevented from termination, in which case termination

1. Process Q is said to be stable when it (a) does not perform any internal transitions, and (b) can always respond to the environment's offer to perform a certain event, given that event is in Q's alphabet.

may never occur.

Event Prefix

The Timed CSP language introduces a more general form of event prefix, which allows the recording of times at which events occur. For example, the traditional CSP expression $a \rightarrow Q$ describes a situation where the system is ready to engage in event a after which it continues as process Q. Timed CSP introduces a slightly different notation for timed event prefixing: $a@u \rightarrow Q$. a@u records the amount of time that has passed since the initial offer of event a till its actual performance, i.e., the amount of time event a was on offer. u is a time variable that can subsequently be used throughout process Q.

Timed and Untimed Timeout

Untimed CSP included the ability of representing timeouts in systems' behavior:

$$Q1 \triangleright Q2$$

Initially the control is with process Q1, but only the performance of the first external event by either Q1 or Q2 will resolve the choice in favour of that process, i.e., Q1 and Q2 may both perform internal events, but they will not resolve the choice. Unfortunately, without any timing information associated with the timeout operator, there is no way to tell when it will happen, leading to nondeterministic resolution of choice.

Timed CSP enhanced the timeout operator to be able to describe time-sensitive process behavior. It is written as $Q1 \overset{d}{\triangleright} Q2$. It specifies a time limit, d, for Q1 to get started. As execution of the system begins, process Q1 is in control. If process Q1 performs any external event before d units of time elapse, then the choice is made in favour of Q1, and Q2 is abandoned without ever being started. Q1 may perform any number of internal events, however, they do not resolve the choice. If by the time d units of time elapse Q1 does not engage in an external event, control is passed to process Q2. If Q1 attempts to engage in a communication event at precisely time d then the result of such action is non-deterministic.

Timeout, being an internal event, brings up the notion of urgency of internal events. This follows from the maximal progress assumption. A process cannot delay performing an internal event and must perform it as soon as it becomes available. “No time may pass whenever an internal event is enabled” [Sch00]. Of course, this is an unrealistic situation in the real world, where time cannot be blocked. This contradictory situation is what actually forces internal events to become urgent. This is simply a mechanism used to express urgency.

Delay

The timeout operator can be used to introduce delays into process descriptions. For example, to delay execution of process Q by d units of time we may construct the following CSP expression:

$$\mathrm{STOP} \overset{d}{\triangleright} \mathrm{Q}$$

Initially the system cannot perform any external or internal events, which is represented by the STOP event. After d units of time, however, the choice is resolved in favor of process Q.

Representation of delays in Timed CSP specifications is quite frequent, therefore, delayed event prefix was introduced with its own notation written as:

$$\mathrm{a} \xrightarrow{d} \mathrm{Q}$$

In the above specification, event a can happen at any point in time. After event a, the process cannot perform any events for d units of time. Only after d units of time can process Q begin its execution. It is important to note that delays following events do not specify precise delays between the events, but rather the minimum delay imposed by the preceding event before the subsequent one can occur.

A waiting process which delays for exactly d time units before successfully terminating can be defined using this construction:

$$\mathsf{WAIT}\ d = \mathsf{STOP} \overset{d}{\triangleright} \mathsf{SKIP}$$

Process WAIT d cannot perform any events for d units of time since STOP cannot perform any events (external or internal). After d units of time, the timeout operator resolves the choice in favor of SKIP which successfully terminates. WAIT 0 and SKIP are semantically equivalent.

The delayed prefix and timeout operators allow us to represent minimum and maximum delays, respectively, between occurrences of events.

Recursion

One of the main differences between the traditional version of CSP and the timed one is the requirement for *time-guarded definitions*. Without such requirement, there is a possibility of performing an infinite number of recursive calls in no time. For example, process $N=a\rightarrow N$ may perform an infinite number of a events in no time. Thus, it is vital to introduce delays which provide a lower bound for the time it takes to reach a fresh recursive call. For example, $N = a \xrightarrow{d} N$. This ensures that d units of time will pass before the next recursive call of N. Of course, the environment may prevent such infinitely fast iterations, however, it is preferable to make such safeguards explicit.

Concurrency

“Concurrent processes must always agree on the performance of evolution transitions. This is due to a combination of the maximal parallelism property and the Newtonian time assumption, which require that concurrent processes are executed together rather than scheduled one after the other, and that time passes at the same rate in all processes” [Sch00]. This is impossible to implement in software, due to processor scheduling, unless every CSP process has a dedicated CPU.

“Two processes can synchronize on an event they both perform only at a time when both are ready” [Sch00]. This may seem trivial but is different from the traditional CSP, where both processes can synchronize on an event as long as both of them have it in their alphabet. Another way to put it is that offers to synchronize can be withdrawn after a specified interval.

The following example is adapted from [Sch00]:

$$\text{HELEN} = (\text{meet} \xrightarrow{30} \text{work} \rightarrow \text{STOP}) \overset{30}{\triangleright} \text{work} \rightarrow \text{STOP}$$

$$\text{CARL} = \text{run\_errands} \xrightarrow{15} \text{SKIP};$$

$$((\text{meet} \xrightarrow{30} \text{home} \rightarrow \text{STOP}) \overset{30}{\triangleright} \text{home} \rightarrow \text{STOP})$$

$$\text{HELEN} \mathbin{||} \text{CARL}$$

HELEN is willing to wait to meet CARL for 30 minutes before giving up and going to work. If they do meet, it will be at least 30 minutes before she goes on to work. Similarly, CARL is willing to wait 30 minutes before going home. CARL is running errands the first 15 minutes that HELEN is available to meet with him, however, it is still possible for them to meet, since times they are both available overlap, i.e., times at which HELEN and CARL are ready to engage in event meet overlap. This is clearly shown in Figure 2-1.

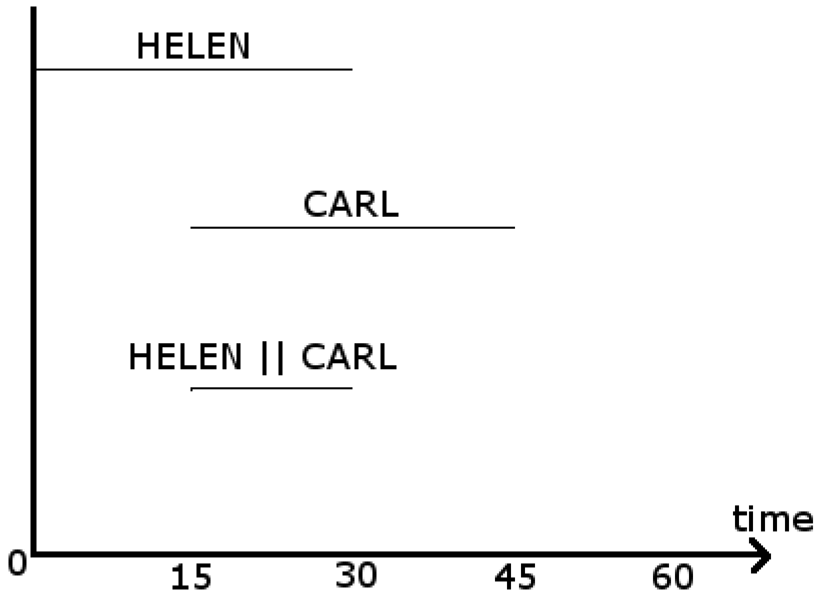

Figure 2-1. Possible meeting time for HELEN and CARL

Now, let’s consider a situation when CARL is not available to meet during the first 45 minutes.

HELEN = (meet $\xrightarrow{30}$ work $\rightarrow$ STOP) $\overset{30}{\triangleright}$ work $\rightarrow$ STOP
CARL = run_errands $\xrightarrow{45}$ SKIP;
    ((meet $\xrightarrow{30}$ home $\rightarrow$ STOP) $\overset{30}{\triangleright}$ home $\rightarrow$ STOP)
HELEN || CARL

By the time CARL is ready to meet, HELEN has already given up and left for work. Even though they both have meet in their common alphabet, they do not synchronize on meet due to their respective timing constraints, as shown in Figure 2-2.

The example of HELEN and CARL illustrates the fact that in order for two processes to synchronize on an event in Timed CSP, not only do the processes need to have overlapping alphabets, but also overlapping times at which synchronization events are offered.

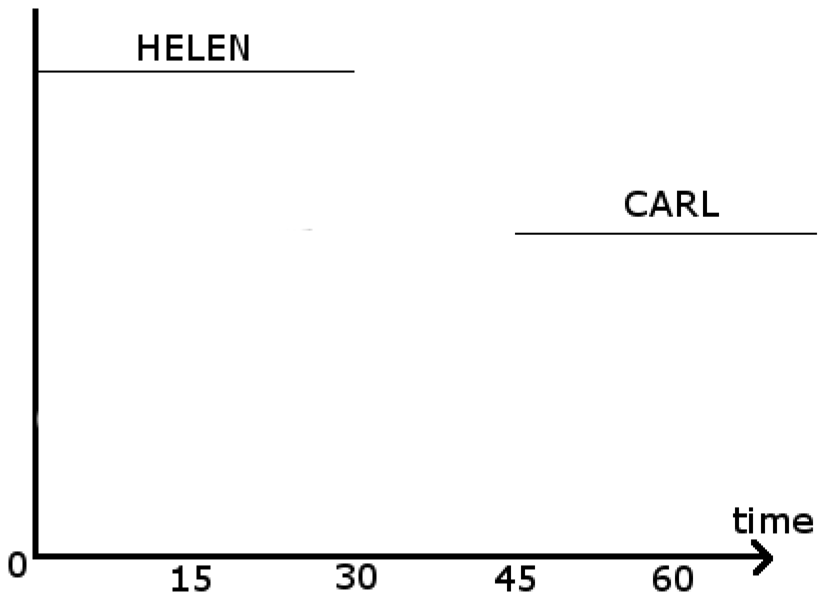

Figure 2-2. Impossibility for HELEN and CARL to meet

Hiding

The property of maximal progress requires an event to occur when all of its participants are ready. This forces hidden events to become urgent, occurring at the instant they are enabled, since all participants of an internal event are identified in the process description.

Renaming

In the timed context, event renaming may introduce nondeterminism. For example, in the following choice situation,

$$(a \rightarrow c \rightarrow STOP) \ \square \ (b \rightarrow c \rightarrow STOP)$$

renaming events a and b to d would result in:

$$(d \rightarrow c \rightarrow STOP) \ \square \ (d \rightarrow c \rightarrow STOP)$$

Here, the left and right sides of the external choice operator become semantically and syntactically equivalent, thus, regardless which side is chosen, the process will remain deterministic.

The same example with the inclusion of time, however, yields nondeterministic results:

$$(a \xrightarrow{3} c \rightarrow STOP) \;\square\; (b \xrightarrow{5} c \rightarrow STOP)$$

When a and b are renamed to d, we get:

$$(d \xrightarrow{3} c \rightarrow STOP) \;\square\; (d \xrightarrow{5} c \rightarrow STOP)$$

Now, choosing the left or right side determines the availability of c, either after 3 seconds or after 5 seconds. Thus, the availability of c after 4 seconds becomes nondeterministic, and external choice becomes equivalent to internal choice:

$$(d \xrightarrow{3} c \rightarrow STOP) \;\sqcap\; (d \xrightarrow{5} c \rightarrow STOP)$$

Interrupt vs. Timed Interrupt

The interrupt construct Q1 Δ Q2 allows the first process Q1 to execute, but it may be interrupted at any time by Q2 executing its first external, i.e., visible event. “Unlike sequential composition, in the interrupt construction both Q1 and Q2 evolve together” [Sch00]. A static interrupting process Q2 which does not evolve as Q1 progresses can be achieved by using a prefix or a prefix choice process as an interrupting process. For instance,

$$Q1 \,\Delta\, e \rightarrow Q2$$

As Q1 progresses, Q2 remains static. Interrupting event e will be offered to the environment at the same time Q1 starts executing. If the environment decides to participate in e, the progress of Q1 will be interrupted and execution will continue with Q2.

There may be situations when a process may not be permitted to run longer than a specific amount of time. In this case, the passage of time itself can set off the interrupt and remove control from the executing process.

$$Q1 \; \Delta_d \; Q2$$

For example, the above specification allows Q1 to run for precisely d units of time, after which control is passed to Q2, unless Q1 has terminated previously. There is no need for Q2 to execute concurrently with Q1 since it will not be invoked until time d.

The important distinction between event-driven interrupt and timed interrupt is that the former might never occur if the environment of the process does not wish to engage in the interrupt event, whereas the latter is bound to occur, since the environment does not participate in the interrupt performance, and thus cannot prevent it from happening.

2.3 Discrete Time Model

The introduction of time into the CSP language described so far concentrated on the continuous time model. This section describes a different approach focused on the discrete timing model where the untimed CSP language is extended with a tock event used to describe the passage of time.

In [Sch00], Schneider describes how to augment CSP with a tock event representing one unit of time and controlling the passage of time in general. There are two alternative interpretations of tock:

1. tock represents evolution of time, and takes time to happen. All events between tock are simultaneous and instantaneous. Consider a mechanical watch, where the seconds hand is constantly moving between every two second gradations. Events happen only on the second marks.
2. tock events are instantaneous and occur every unit of time. Now, consider a digital watch, where a second hand moves from one second’s gradation to another, but does it almost instantaneously, like a drum beat.

Thus, the introduction of the tock event into the untimed CSP language brings up a non-continuous view of time, that is, time is not allowed to pass unless a tock event occurs.

There are two approaches as to how a tock can be used:

- tock can be a simple addition to the untimed CSP language.
- tock can be linked to the Timed CSP language.

2.3.1 *tock*-enhanced "Traditional" CSP

Introduction of a timed event into the untimed language alphabet has several implications as to how time is treated. Processes are no longer indifferent to the passage of time as in standard CSP. tock is the only medium that allows time to pass. For example, a simple specification,

$$Q = a \rightarrow b \rightarrow STOP$$

if viewed in the timed context (even though tock is not included), does not mean process Q is indifferent to time. On the contrary, events a, b, and STOP must be performed before any time can pass. Furthermore, after execution of STOP no time can pass, since STOP deadlocks and is not able to perform any events. This leads to the situation where the notion of time has to be included explicitly, for instance:

$$Q = a \rightarrow tock \rightarrow b \rightarrow tock \rightarrow STOP$$

Explicit declaration of tock events becomes even more important during specification of recursive processes. Without tock, processes would perform an infinite number of events in no time. Thus, to ensure time-guardedness, recursive processes must include at least one tock event.

$$Q = a \rightarrow tock \rightarrow Q$$

2.3.2 *tock* and Timed CSP

Timed CSP can be combined with *tock*-enhanced standard CSP by means of translating specifications written in the timed language to tock event-driven processes. Such translation would require modeling all timed CSP constructs (timeout, delay, and interrupt) using tock events. Unfortunately, the translation process introduces certain challenges as the same operators may be treated differently in the timed and untimed models. For example, passage of time should not resolve external choice in timed CSP, however, when translated to traditional CSP, external choice may end up with tock events on either side, meaning simple passage of time will resolve choice. Moreover, tocks on either side of external choice may introduce nondeterminism. This situation may be fixed by resolving the choice considering events following tock. However, such “fixes” introduce unnecessary complexity into the system specification process.

After reviewing Timed CSP and *tock*-CSP, we had to make a decision as to which timing model would be more appropriate for CSP++ and software synthesis. When all pros and cons were considered, the decision was made to implement Timed CSP constructs in CSP++. Chapter 3 provides more information as to why Timed CSP was a better solution for CSP++.

2.4 Verification of Timed CSP Specifications

Formal Systems, Ltd. is currently the only company that markets formal verification tools for Communicating Sequential Processes:

- ***Checker*** is the simplest of the tools, and is used to verify the correctness of CSPm syntax in a given specification.

- ***Process Behaviour Explorer*** (ProBE) is used to the explore the possible states, or execution paths, of all processes in a given specification.
- ***Failure-Divergence Refinement 2*** (FDR2) is the most sophisticated tool and is used to verify the absence of deadlocks, livelocks, or any nondeterministic properties in a specified system, or to expose such properties and, if possible, provide counterexamples.

The three tools listed above provide the full range of support for writing and verifying an untimed CSP specification. Unfortunately, if one decides to write a Timed CSP specification, the tools cannot be used as they only support the untimed CSP operators.

Up until recently, there were no verification tools for Timed CSP specifications. In "Reasoning About Timed CSP Models," Dong, Zhang et al. [DZSH06] reported on the development of an interactive system used to compose and reason about Timed CSP specifications. Their HORAE system uses Constraint Logic Programming (CLP) as the underlying verification framework. The front-end of the tool is implemented in Java and has the following features:

- building Timed CSP models
- specifying properties in a systematic way
- verifying various kinds of properties with counterexamples provided, if any
- generating LaTeX presentation of the Timed CSP models

The system incorporates an editor used to write Timed CSP specifications. One of the underlying components of the system, tcsp2clp, then translates the Timed CSP specification to a CLP specification, which can be verified for safety and liveness, timewise refinement, and variable bounds. Several popular Timed CSP case studies including Timed Vending Machine, Dining Philosophers, and Railroad Crossing have been used to test the system (these case studies were also implemented in CSP++ and the results are presented in Appendix A). The results matched those obtained from FDR2. It should be noted that unlike HORAE, FDR2 does not have the ability to verify the timing properties of specifications, and, thus, only results which abstract from the notion of time were compared.

2.5 Overview of CSP++

This section provides an overview of CSP++ including a brief outline of changes the tool has seen and a more technical description of the system itself. This is done to help the reader understand the challenges encountered when extending CSP++ with Timed CSP operators.

2.5.1 Summary of CSP++ development changes

CSP++ [Gar00] [Gar03] was originally created as part of W. Gardner's Ph.D. thesis work at the University of Victoria and later updated and extended by Stephen Doxsee, as part of his master's thesis [Dox05], and by Joshua Moore-Oliva.

2.5.1.1 csp12 to CSPm

At first, CSP++ accepted a local machine-readable dialect of Communicating Sequential Processes, called csp12. This was done to conform to an in-house verification tool developed by Dr. M.H.M. Cheng at the Department of Computer Science, University of Victoria, BC. The system was later reengineered by Stephen Doxsee at the University of Guelph to accept CSPm syntax [Dox05, DG05]. The move to CSPm syntax allowed inclusion of new features, not previously available with csp12. This also allowed specifications to be directly verifiable by the robust commercial tool FDR2. The main relics remaining from csp12 are the names of classes: `Agent` class which implements CSP process semantics and `Action` class which implements CSP event semantics.

2.5.1.2 Threading libraries

CSP++ was originally based on AT&T’s USL (Unix Systems Library) task library. By the late 90’s these non-preemptive, non-prioritized coroutines had become obsolete, and the decision was made to move to preemtable, kernel-level LinuxThreads. CSP++ was demonstrated to work with the new threading library, but “because the original CSP++ design was based on non-preemptive threads, there was always a suspicion that the move to preemtable threads was done too hastily and without complete identification of critical sections in the framework code” [Dox05]. CSP++ was once again ported to use the non-preemptive threading library, GNU Pth, which is being used to date.

2.5.1.3 General CSP++ changes

A number of changes to CSP++ were also contributed by Joshua Moore-Oliva, then an undergraduate student at the University of Guelph.

Removing the framework’s reliance on heavyweight C++ I/O classes and Standard Template Library (STL) reduced the “footprint” of CSP++ at run-time. The reengineered version, or “micro edition,” became more suitable for targeting embedded systems with limited hardware resources.

A suite of regression tests based on the CppUnit Framework and the Boost Test Library was developed. This suite is constantly being updated and is utilized every time a change to the system is made to insure that no features were broken in the meantime.

CSP++ code was also reorganized to allow installation via autoconf. This made porting to any Unix/Linux platform much easier.

2.5.2 CSP++ Design Flow

CSP++ is an object-oriented application framework which allows CSP specifications to be directly executable and extensible, while retaining the original specification properties that may have been verified with FDR2 to ensure the absence of deadlocks and livelocks, and deterministic character of the specified system [Gar03].

C++ code synthesis from CSPm specifications is achieved by running the specification through the cspt translator (CSP++ front-end) and then linking the generated code to the system's runtime library. Running the generated code will print the trace of the system, a list of executed events. At this stage these "events" remain abstract implementations of events described in the CSPm specification, and the whole system is very suitable for simulation purposes, but not much else. What sets CSP++ apart from other software synthesis tools is the ability to extend these abstract events via user-coded functions (UCFs). The tool provides the ability to formally specify and subsequently generate C++ code for only critical parts (responsible for process synchronization and interaction) of software products being designed. Furthermore, parts of the system responsible for simple input/output operations or any calculations can be plugged into the formal specification via UCFs, which saves developers from needless efforts to specify those parts formally. Such freedom over which parts of the system need to be formally specified was subsequently dubbed "selective formalism" [Gar03].

The whole process of running a CSPm specification through CSP++ and extending it with UCFs is illustrated in Figure 2-3.

A developer starts with a CSPm specification, which is run through verification tools such as FDR2 and ProBe until the specified system is free of deadlocks and livelocks,

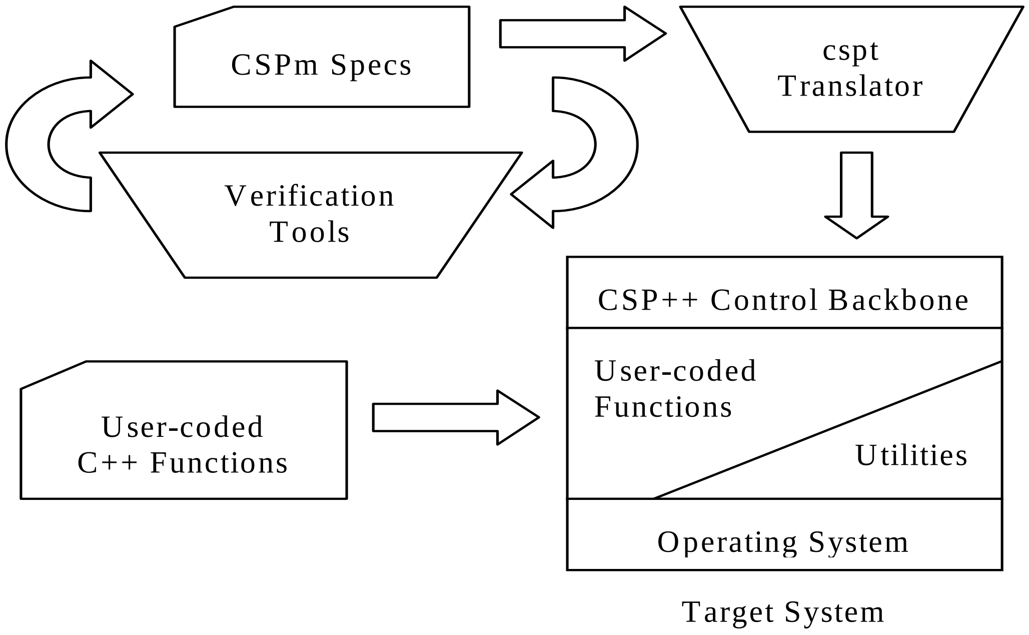

Figure 2-3. CSP++ Design Flow

is deterministic and has other desired properties. The *cspt* translator then translates the CSPm specification into C++ source code. The source code is compiled and linked with the CSP++ run-time library, at which point the synthesized program is ready to be executed. The executable specification can be extended with user-coded functions which are linked to CSP events included in the specification. As long as UCFs do not participate in any inter-process communication or synchronization, system properties verified with FDR2 and ProBe will stay intact.

2.5.3 CSP++ class hierarchy

CSP++'s back-end is an object-oriented application framework consisting of several key classes, shown using UML in Figure 2-4. While adding Timed CSP constructs to CSP++, some of these classes were changed or extended with additional features. To better understand implementation details presented in the next chapter, we present a brief overview of the principal CSP++ framework classes. A more detailed description of CSP++ framework classes can be found in [Gar00].

- **Agent**—"embodies a process definition ... representing a schedulable thread of control." Every CSP process definition is represented by an individual function of type `AgentProc`. An Agent class constructor schedules execution of its `AgentProc`s. When one AgentProc finishes its execution, it may designate another `AgentProc` to be executed after it until the CSP SKIP process is encountered, which signifies the end of the current process's execution. Throughout this project a number of functions were added to the `Agent` class to handle the logistics of the newly-added CSP operators.
- **Action**—represents the two types of CSP events: channels, which pass data, and atomic, which do not.

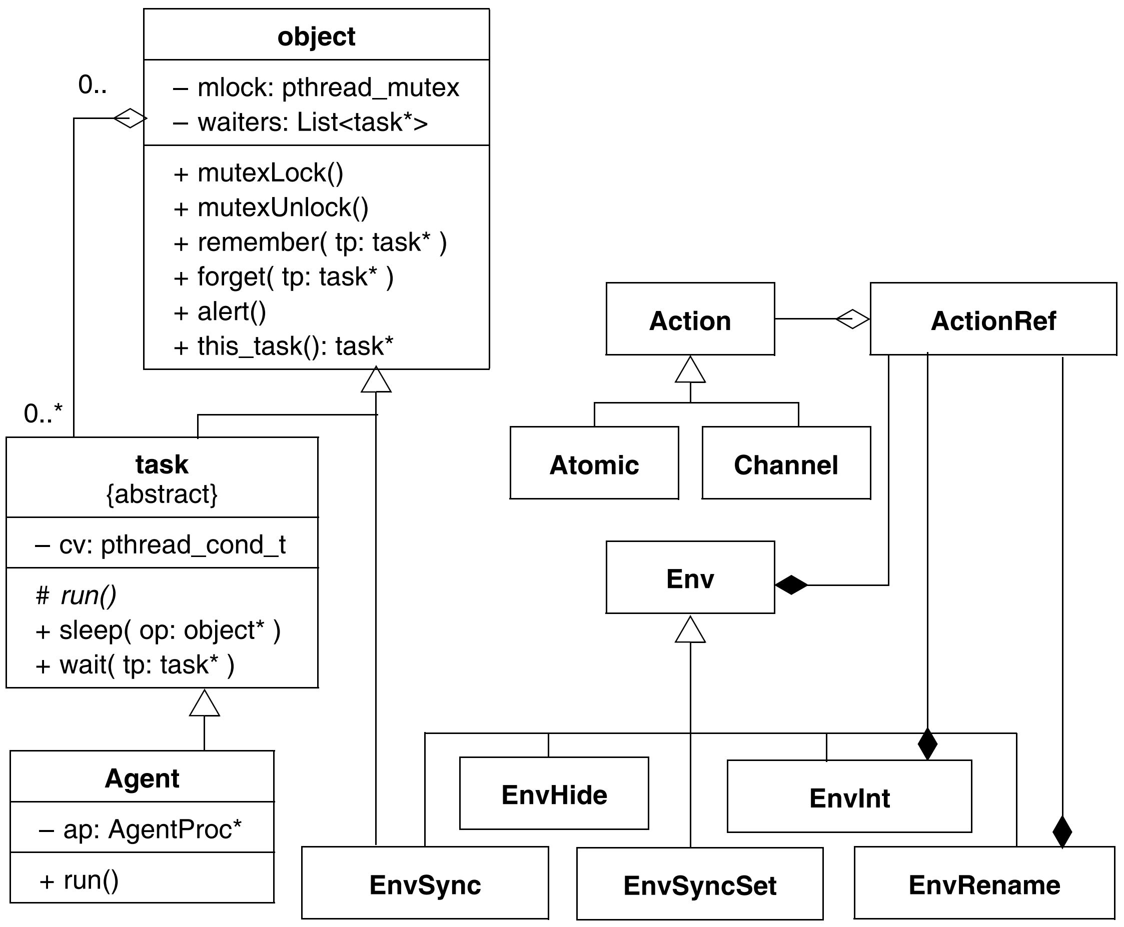

Figure 2-4. CSP++ Class Diagram

- **Env—**declarative objects that introduce an Action reference (`ActionRef`) into a process's environment to accommodate synchronization, event hiding, event renaming, and, finally, interrupts, introduced as part of this thesis work. `EnvInt` is a subclass of `Env` that implements interrupts.

Detailed discussion of how these classes are utilized and cooperate is given in the next chapter.

2.6 Survey of competing CSP libraries

Communicating Sequential Processes was never intended to be a programming language, but does provide the necessary features to describe complex parallel systems. A number of projects have been developed that tried to bridge CSP with conventional programming languages. [DG05] briefly describes the three main approaches taken to marry CSP with a full-features programming language:

- *Programming languages inspired by CSP.* Examples of such programming languages include LOTOS and occam, and even though these languages cannot be directly formally verified, they provide means of doing so through intermediate steps. For instance, LOTOS has CADP [FGM+92], a toolset supporting verification, while [HJ95] provides steps for translating a CSP specification into an occam program.
- *Formal libraries.* A number of projects have focused on the development of libraries of classes and functions that follow formal semantics and are tailored towards a particular programming language. A number of libraries originated at the Computing Laboratory at the University of Kent, England: C++CSP for C++ [BW03, Bro07a], JCSP for Java [WBM+07], CCSP for C, and xCSP for a number of languages. xCSP spun off as a separate commercial project developed by Quickstone, but has since disappeared. Another library aimed at the Microsoft .NET framework, CSP.NET [Ols06, LO06], was developed at the Department of Computer Science at the University of Copenhagen, Denmark. The project has since become known as Jibu, maintained by a Department of

Computer Science spin-off company Axon7, and currently features a uniform API for the .NET Framework, C++, and Java [Axo07].

Such formal libraries simply provide alternative concurrency models, and usually do not require any knowledge of CSP, unless the original project specification was given in CSP. The upside to this is that any programmer can take advantage of such libraries invoking calls to functions, which implement particular CSP constructs, provided in the library API. The downside, however, is that no direct formal verification is possible. Moreover, if the original project specification was written in CSP, much efforts will be put into hand-translating such specifications into programming code linked to the library function calls.

- *Direct code synthesis* features direct code generation from CSP specifications. Raju et al. [RRS03] describe a tool featuring direct translation of CSPm specifications to Java and C programs linked to JCSP and CCSP libraries. The paper dates to 2003 and no additional developments have been reported since then. CSP++ falls into this category as well, but provides more comprehensive support for CSPm [DG05].

Since there has not been any apparent news about the automated translator for JCSP and CCSP described above, in Chapter 5 we will present a more detailed comparison between CSP++, C++CSP and JCSP libraries, describing variety of features, pros and cons, and implementation scenarios.

3 CSP++ and Timing Operators

The original design of CSP++ did not feature any timing operators, which limited the scope of system design specifications that could be synthesized with our tool. Once the decision was made to extend the existing CSP++ with timing constructs, we were left with two options: (1) use the discrete timing model with tock, or (2) extend the translator and the OOAF with additional timing operators, such as timed prefix, timeout, and interrupt.

The first option of using tock alongside standard CSP did have its advantages, but in the end was rejected in favour of Timed CSP. The main advantage of *tock*-CSP was the minimum number of changes required to CSP++ in order to accommodate this timing paradigm. In fact, CSP++ is currently able to handle *tock*-CSP. Required changes would have included:

- automatic detection of tock in all synchronizing processes, and
- linking tock to a function that would handle different time units and suspend the current thread for the specified amount of time.

Despite the relative ease of tock implementation, the disadvantages of using it were significant. There were two views of tock described in Section 2.3: (1) tock represents a time evolution and takes one unit of time to happen, and (2) tock is like a drum beat, instantaneous, but happening every unit of time. All other events between two consecutive tocks in a given specification are simultaneous and instantaneous. The first interpretation of tock is not realistic for CSP++ since there is no true simultaneity, and the CPU would be constantly evolving rather than making progress. Furthermore, UCFs, or any other events, are not truly instantaneous and must take time to occur. The second tock model is more realistic, but brings up a problem of UCFs taking more than one tock (one time unit) to execute. This would delay the next tock synchronization between processes leading to a deadlock situation and threatening the formal properties of a CSP specification. Moreover, *tock*-CSP specifications may grow extremely long with large numbers of tocks, which leads to tedious work and hard-to-understand specifications. Finally, the biggest concern with using *tock*-CSP in CSP++ was the fact that, even if technical and esthetical challenges were overlooked, use of tock only solved one problem, *representation of timed delays*. Such constructs as timeouts and interrupts would still need to be added to grasp the full range of timed operators.

In the end, *tock*-CSP did not seem to be the best fit for CSP++, and the decision was made to implement Timed CSP operators.

This chapter describes the work of extending CSP++ with timing operators: timed prefix, timeouts, and interrupts. We begin the chapter by describing the theoretical issues that had to be resolved to avoid significant system reengineering while keeping it faithful to CSP semantics. The chapter continues with the description of changes that had to be made to the CSP++ translator, cspt, and the runtime library to accommodate the new timed operators.

3.1 Theoretical Issues

In this section we will discuss theoretical and technical difficulties encountered during this research. We present reasons for the sacrifices and trade-offs that had to be made to make

Timed CSP++ a useful software synthesis tool.

3.1.1 Timing Postulates

The introduction of Timed CSP was accompanied by a number of assumptions, or timing postulates, that governed the behavior of new timing operators [Sch00]. When we set out to extend CSP++ with timing operators, we had to carefully consider each postulate and decide what implications it may hold if left unimplemented. Below is the list of timing assumptions, presented earlier in Section 2.2.2, and our considerations:

- ***Instantaneous events***. Events are performed at an instance the process is ready to perform them and take no time to execute. Events that take time to complete may be modeled as an event at which action begins and an event at which it ends.

The universe is bound by the laws of physics, meaning nothing in our world is instantaneous; hence, this postulate is impossible to implement. “The treatment of events as synchronizations means that it is appropriate to consider their occurrence as instantaneous, performed by a process at the precise point it becomes committed to the event. CSP is designed to consider systems in terms of synchronizing processes, and so the treatment of events as instantaneous naturally follows” [Sch00]. In reality, every computer program is based around CPU availability, so, realistically, it will take time for a computer to execute an event even when a process is committed to it. The impossibility of implementing this postulate does not break the flow of time, as in Timed CSP delays are treated as time intervals occurring between finishing one event and beginning the next, and this property maps directly to our CSP++ implementation. Therefore, we are only concerned with intervals between events, and not the amount of time one event takes to execute.

When a Timed CSP specification is synthesized and corresponding C++ code is generated, CSP++ does not produce a timed execution trace, but rather an untimed one. If one decides to carry out a timed analysis, the trace produced by CSP++ will not match the theoretical timed trace, as events do take time to execute. The consequence of this is that a designer must keep in mind the fact that events take time to execute.

- ***Newtonian time***. "Time passes with reference to a single conceptual global clock"[Sch00].

CPU cycles can be treated as the single conceptual global clock, therefore, this postulate directly translates to the "machine" world.

- ***Real-time***. Time can only be represented in terms of real non-negative numbers.

CSP++ accepts time specifications in the form of non-negative integers, while the numbers themselves can represent different time units: milliseconds (ms), seconds (s), and minutes (m). Time units are described in more detail in Appendix B. These time units can be coded in the specification file or be changed via command-line flags before actual execution, with the command-line flags taking precedence over specification units. This is done to ease the simulation and debugging of a simulated program which may take a long time to execute otherwise.

- ***Maximal parallelism*** represents the notion of each process having its own processor that it executes on. Any scheduling analysis must be made explicit.

The current version of CSP++ can only take advantage of a single processor, in part due to its underlying threading library, GNU Pth. Hence, every process, represented by a Pth thread, is subject to Pth scheduling policy, which incorporates thread aging to avoid starvation. Use of multithreading is sufficient in an attempt to achieve at least the illusion of process parallelism, but without having at minimum one CPU available per active CSP process, maximal parallelism cannot be achieved. A system designer must keep this fact in mind when constructing a CSP specification: every CSP process will be mapped onto a separate Pth thread.

- ***Maximal progress***. An event must occur at an instant all synchronizing parties are ready to perform it. This assumption also includes urgency of internal events. They must be performed at an instance they become available.

In CSP++ event synchronization occurs when all synchronizing processes are ready to perform it. A process arriving at a synchronization point checks if it is the last one to arrive. If so, synchronization occurs and processes continue their separate ways. If not, the process will raise the appropriate flag in the framework and go to sleep, to be later woken up by the last arriving process. This satisfies the requirement for all synchronizing parties to perform synchronization together, however, it doesn't happen instantaneously, but takes some time for the CPU to perform the appropriate operations.

In the sense of maximal progress, internal events are treated as regular events, and are executed by the running process when are encountered, with the only exception being that they do not show up in the trace.

- ***Finite variability***. Only a finite number of events can be performed in a finite amount of time.

Being bound by the laws of physics, CSP++ fully satisfies this requirement.

- ***Synchronous communication***. Synchronization events require simultaneous participation of all involved parties.

Multiple CPUs would be required to come close to simultaneous synchronization of all participating parties, with every process running on a dedicated CPU. CSP++ utilizes the GNU Pth threading library that cannot take advantage of multiprocessor systems, and thus, every running thread is a subject to the Pth scheduler. Nonetheless, inability to satisfy this requirement does not change the behaviour of a simulated system, as every synchronizing party is unable to progress beyond a synchronization point until it is crossed by every participating process.

After reviewing all of the Timed CSP postulates, it became apparent that two of them, instantaneous events and maximal parallelism, cannot be implemented in CSP++. This means that CSP++ is suitable for modeling and synthesis of soft real-time systems, where only minimum timing guarantees can be met. For example, based on the underlying GNU Pth threading library, we can guarantee that in the following sample specification:

$$\mathsf{SAMPLE} = \mathsf{a} \xrightarrow{5} \mathsf{b} \rightarrow \mathsf{SKIP}$$

at least five seconds (if time units were set to seconds in the specification) will pass after a finishes execution and the CSP++ framework attempts to engage in b.

However, the main barrier to using CSP++ for hard real-time systems—those whose correctness or even safety depends on responding to events within tiny latencies (e.g., on the order of milliseconds or less)—is not the above postulates, but rather the lack of timing constraints in Timed CSP. There are no constructs to specify that, for instance, event b *must* happen within *n* time units of a previous event. And then, even if there were, a software synthesis tool ought to generate a custom schedule guaranteed to satisfy the constraints, but this is contrary to the way in which the CSP++ framework is currently designed. It does not generate a schedule at all, but leaves the underlying threads package to carry out non-preemptive scheduling according to its own algorithm. For such reasons, CSP++, even enhanced with Timed CSP operators, does not aspire to be a tool for synthesizing hard real-time systems.

3.1.2 GNU Pth threading library and timing operators

GNU Pth is a highly portable, POSIX-compatible, non-preemptible, priority-based threading library. Use of Pth had its implications on the implementation of Timed CSP in CSP++.

The most significant challenge posed by Pth threads was the fact that the library is non-preemptible. This means that once a particular thread of execution gets control of the CPU, it will only release it if a blocking situation occurs, such as input/output or waiting for synchronization, or the thread will explicitly yield its control in a cooperative way. The non-preemptive nature of Pth threads came greatly into play during implementation of the interrupt operators.

To recall the semantics of interrupts, let us consider the following example:

HOUSE_CLEANING Δ phone_call → PHONE_CONVERSATION

Initially, as the execution begins, process HOUSE_CLEANING has control of the CPU. At any given time a phone may ring, and house cleaning duties have to be **immediately** abandoned in favour of PHONE_CONVERSATION. The non-preemptive nature of Pth threads makes this requirement unimplementable in practice. With CSP++, HOUSE_CLEANING will run to completion unless the process contains a blocking call which would make it release control of the CPU and give phone_call a chance to run.

Consider these alternate scenarios:

HOUSE_CLEANING = clean_bathroom → wipe_dust →
clean_dishes → SKIP

PHONE_CONVERSATION = hello $\rightarrow$ ok $\rightarrow$ good_bye $\rightarrow$ SKIP

With Pth threads, HOUSE_CLEANING will run to completion, because there are no blocking calls that would put it to sleep.

HOUSE_CLEANING′ = clean_bathroom $\xrightarrow{5}$ wipe_dust $\xrightarrow{5}$ clean_dishes $\rightarrow$ SKIP

PHONE_CONVERSATION = hello $\rightarrow$ ok $\rightarrow$ good_bye $\rightarrow$ SKIP

In the second scenario, HOUSE_CLEANING′ blocks (sleeps) twice during its execution. At those moments HOUSE_CLEANING′ will release control of the CPU and will give phone_call a chance to happen. If phone_call were to happen between clean_bathroom and wipe_dust, execution will continue with PHONE_CONVERSATION and will result in the following system trace <clean_bathroom, phone_call, hello, ok, good_bye>. A more detailed discussion on the implementation of the interrupt operators is presented later in this chapter.

3.1.3 Untimed Timeout—a non-deterministic CSP operator

When considering what timed operators to implement in CSP++, it was also decided to extend the tool with two untimed CSP operators, untimed timeout and untimed interrupt, as their semantics are fairly close to their timed counterparts, minus the time.

Unfortunately, the untimed version of the timeout operator introduces non-determinism to system design. For instance (the example is adapted from [Sch00]),

OFFER = (cheap $\rightarrow$ SKIP) $\triangleright$ (lapse $\rightarrow$ standard $\rightarrow$ SKIP)

BUYER = cheap $\rightarrow$ SKIP

OFFER $\|_{\text{cheap}}$ BUYER

An offer is available at a special price for some time, after which the offer lapses and the price goes back to standard. A buyer tries to make a purchase at the special offer price, represented by synchronization of BUYER and OFFER on the cheap event. With timing absent, there is no way to tell when the offer will lapse, making resolution of the timeout situation an internal decision. Therefore, there is no way to tell whether at the time of synchronization OFFER's cheap event will be available and the transaction will complete successfully, or the system will deadlock as BUYER tries to buy for cheap, while only standard price is available.

Non-deterministic flow of control is normally undesirable in software synthesis, hence, we did not want to implement the untimed timeout in its original form in CSP++, but still believed it could be a useful operator if interpreted in a deterministic way. The decision was made to make the untimed timeout a polling operator. The CSP++ framework will poll on the left hand side of the timeout operator to see if the process is ready to engage in it right away, which will determine the flow of control. Considering the above example, when process OFFER gets control of the CPU, the framework would check if OFFER's event was ready and able to engage in the cheap event. If so, cheap will be performed leaving the following trace <cheap>; if not, the timeout will occur, giving the trace <lapse, standard>. This change avoided introducing non-determinism into CSP++ while keeping system design semantics unaltered.

3.2 CSP++ Changes

The first implementation step was the extension and modification of the CSP++'s front-end, the cspt translator, which accepts a CSPm specification and creates a parse tree used to generate the corresponding C++ objects. The translator is based on Flex (The Fast Lexical Analyzer) and Bison, a general-purpose parser.

Extending the translator was a two-step process: first, we had to add recognition of the new Timed CSP operators and grammar rules, and, second, add the corresponding classes to generate C++ code from parsed-out objects.

3.2.1 Overview of the translator

When given a CSPm specification, the Flex part of the translator first parses the specification, splitting it into tokens, and then passes it to Bison, where the tokens are combined using BNF-like grammar rules. Grammar rules either create new `ParseNode` objects, pushing them onto an object-oriented parse tree, or add a token to an operand preparing to create a new `ParseNode` object. Created objects will be of the `ParseNode` subclasses: `PNcop` for complex operators, `PNtok` for simple tokens, and `PNcid` for complex identifiers.

During the code generation stage, the translator goes through the created parse tree, and executes `prep()` and `gen()` functions for every `ParseNode` object. `prep()` is responsible for preparation of `ParseNode` objects for later generation, and may not be needed by all `ParseNode` objects. `gen()` is responsible for actual C++ code generation. Both functions can be overridden to fit particular needs of every `ParseNode` object.

The translation process is described in greater detail in [Gar00]. Table 3-1 shows the BNF syntax for the CSPm operators currently supported by CSP++. It was updated in [Dox05], and is now updated again with Timed CSP operators. Generated code for each new operator is shown in tables starting from Section 3.2.3.

Table 3-1. BNF syntax with corresponding parse node classes

Accepted CSPm syntax in BNF	Parent			Subclass Name	Pseudocode[a]		
	PNtok	PNcop	PNcid		ctor[b]	prep()	gen() and details for entries marked ">" (ctor = constructor)
				`ParseNode`	>	OK	ctor: store line number gen(): OK
	*			`PNtok`	{ }	-	-
		*		`PNcop`	{ }	apply prep/gen to each operand in turn; stop on bad status	
			*	`PNcid`	{ }	>	prep(): apply to each arg/subscript; stop on bad status gen(): output name
<definition> ::= <signature> '=' <agent>		*		`PNdefn`	{ }	NC	prep signature and agent; use agent's symbol entry to gen AGENTPROC, arg #defines, and FreeVars (genAgentProc); gen agent body; "ENDAGENT" if needed; gen arg #undefs (genEndAgent)
| <agent> '[]' <agent> { '[]' <agent> }		*		`PNchoice`	>	-	ctor: continue only if all agents are prefix"Agent::startDChoice(n)"; set flag for PNinput (DatumVar gen); genPre actions;"Agent::whichDChoice()"; genPost agents
<prefix> ::= <action> <delay> <agent>		*		`PNtimed-prefix`	{ }	-	gen(): - genPre(): gen action genTime(): gen sleep time genPost(): gen agent
<delay> ::= '-' NUM '->'	*			`PNdelay`	{ }	-	gen(): "Agent::nap(time):"
| <agent> '[>' <agent>		*		`PNtimeout`	>	-	ctor: continue only if the left hand side agent is prefix or timedprefix "Agent::startDChoice(1)"; set flag for PNinput (DatumVar gen); genPre actions; "Agent::whichUTChoice()"; genPost agents
| <agent> '[' NUM '>' <agent>		*		`PNtimed-timeout`	>	-	ctor: continue only if the left hand side agent is prefix or timedprefix "Agent::startDChoice(1)"; set flag for PNinput (DatumVar gen); genPre actions; "Agent::whichTTChoice(time)"; genPost agents

Table 3-1. BNF syntax with corresponding parse node classes (Continued)

Accepted CSPm syntax in BNF	Parent			Subclass Name	Pseudocode[a]		
	PNtok	PNcop	PNcid		ctor[b]	prep()	**gen()** and details for entries marked “>” (ctor = constructor)
| <agent> '/\\' <agent>		*		`PNinterrupt`	>	-	ctor: continue only if the right hand side agent is prefix or timedprefix "Agent::startDChoice(1)"; genPre actions;"Agent::whichUIChoice()"; genPost agent; start left hand side agent; push interrupt env. obj onto stack, “Agent::startUI()”; change thread priority; “WAIT” agent
| <agent> '/' NUM '\\' <agent>		*		`PNtimed-interrupt`	{ }	-	"Agent::startDChoice(1)"; genPre actions;"Agent::whichTIChoice()"; genPost agent; start left hand side agent; push interrupt env. obj onto stack, “Agent::startUI(time)”; change thread priority; “WAIT” agent
| <agent> ‘;’ <agent> {‘;’ <agent>}		*		`PNseq`	{ }	-	gen each agent, flagging last one
| <agent> ‘||’ <agent> | <agent> ‘|||’ <agent>		*		`PNcompose`	{ }	>	prep(): prep simple agents; complex: use agent’s symbol entry to extract subagents (makeSubAgent), then gen gen(): “Agent::compose(*n*)”; “START” each agent; “WAIT” each agent
| <agent> ‘^’ ‘{’ <name>{,<name>} ‘}’ | <agent> ‘\\’ ‘{’ <name>{,<name>} ‘}’ | <agent> ‘#’ ‘{’ <rename>{,<rename>} ‘}’		*		`PNenv`	{ }	-	gen the ActionRefs; “.”; “sync()”, “hide()”, or gen PNrename; gen the associated agent; “Agent::popEnv(*n*)” if needed
| <agent> ‘+’ <agent>		*		`PNor`	{ }	-	gen each agent
| STOP	*			`PNstop`	{ }	-	“Agent::stop()”
| SKIP	*			`PNskip`	{ }	-	set flag to get ENDAGENT generated
| <UID>[‘(‘ <exp>{,<exp>} ‘)’]			*	`PNconst`	{ }	>	prep(): find in agentTable, get agentproc name via bindSig(*args*) gen(): “CHAIN”, “START”, or “START/WAIT” depending on context

Table 3-1. BNF syntax with corresponding parse node classes (Continued)

Accepted CSPm syntax in BNF	Parent			Subclass Name	Pseudocode[a]		
	PNtok	PNcop	PNcid		ctor[b]	prep()	gen() and details for entries marked ">" (ctor = constructor)
| IF <exp> THEN <agent>)		*		`PNifthen`	{ }	>	prep(): prep agent gen(): "if ("; gen exp; ") {"; gen agent; "}"
<prefix> ::= <action> '->' <agent>		*		`PNprefix`	{ }	-	gen(): - genPre(): gen action genPost(): gen agent
<signature> ::= <UID> '(' <numvar>{,<numvar>} ')'			*	`PNsig`	>	>	ctor: find in agentTable, or insert new variant prep(): find signature in agentTable, set its symbol entry as the translation context; setup symbol entry to handle symbols for variant (prep) gen(): NC
<numvar> ::= (<NUM>	*			`PNnum`	{ }	-	output value
| <VAR>)	*			`PNvar`	{ }	>	prep(): report to agent's symbol entry (addvar) with "global" flag if in subagent gen(): output var name, maybe globalized, obtained from agent's symbol entry (ref)
<action> ::= (DONE	*			`PNdone`	{ }	-	-
| <LID>['(' <exp>{,<exp>} ')']			*	`PNatomic`	>	OK	ctor: find/insert in actionTable gen(): output name, gen subscripts
| <LID> '?' (<VAR> | <datumvar>)		*		`PNinput`	>	-	ctor: new PNchannel gen(): if datumvar, "DatumVar" *temp* "=" gen datumvar; gen PNchannel; ">>"; gen PNvar or *temp*
| <LID> '!' <exp>)		*		`PNoutput`	>	OK	ctor: new PNchannel gen(): gen PNchannel; "<< ("; gen exp; ")"
	*			`PNchannel`	>	-	ctor: find/insert in actionTable gen(): output name

Table 3-1. BNF syntax with corresponding parse node classes (Continued)

Accepted CSPm syntax in BNF	Parent			Subclass Name	Pseudocode[a]		
	PNtok	PNcop	PNcid		ctor[b]	prep()	**gen()** and details for entries marked “>” (ctor = constructor)
<datumvar> ::= <LID>[‘(’ <VAR>{,<VAR>} ‘)’]			*	`PNdatumvar`	>	-	ctor: find/insert in datumTable gen(): output name, gen subscripts
<name> ::= <LID>[‘(’ <NUM>{,<NUM>} ‘)’]			*	`PNaction`	{ }	OK	find in actionTable, output ActionRef, gen subscripts
<rename> ::= <name> ‘=’ <name>		*		`PNrename`	>	OK	ctor: get 2nd name into actionTable (makeAtomic) gen(): gen 1st PNaction; “.rename(”; gen 2nd PNaction; “)”
<exp> ::= (<numvar>	*see <numvar> above*						
| <LID> ‘(‘ <exp>{,<exp>} ‘)’			*	`PNdatum`	>	OK	ctor: find/insert in datumTable gen(): output name, gen subscripts
| ‘–’ <exp> | <exp> <op> <exp>) <op> ::= (‘+’ | ‘–’ | ‘*’ | ‘/’ | ‘=’ | ‘<’ | ‘>’ | ‘=<’ | ‘=>’ | ‘<>’)		*		`PNop`	{ }	OK	“(“; gen left exp; *op*; gen right exp; “)”
Prep-time node substitution: new extracted subagent’s <signature>			*	`PNsigSub`	{ }	>	prep(): note subagent no. in translation context gen(): NC
replaces complex <agent> subtree, refers to subagent	`PNconst`			`PNconstSub`	{ }	OK	default to PNconst::gen()

a. *Pseudocode abbreviations:* { } = no-op; - = default to parent’s method; OK = no-op, return good status (0); NC = method is not called; “foo” = output “foo”
b. *Constructor note:* The obvious action of storing arguments in data members is not explicitly written out.

3.2.2 Operator precedence rules

Special arrangements with Formal Systems, Ltd., made their FDR2 Flex and Bison files available for our use. This allowed us to align cspt operator precedence rules to comply with FDR2, thus further ensuring the compatibility of CSP++ with FDR2. Table 3-2 lists the syntax and precedence of operators available in FDR2 and CSP++. Some of the CSPm operators are not available in CSP++, including an additional unlisted category called “replicated” operators. The latter are a shorthand way of expressing multiple similar instances

Table 3-2. FDR2 and CSP++ operator precedence
strongest (top) to weakest (bottom)

Category	**Description**	**FDR2 operators**	**Associativity**	**CSP++ operators**	**Associativity**
Application	function application	f (0)			
	renaming	[[<-]]		[[<-]]	
Arithmetic	unary minus	-		-	
	multiplication	* / %	left	* /	left
	addition	+ -	left	+ -	left
Sequence	catenation	^			
	length	#			
Comparisons	ordering	< > <= >=	none	< > <= >=	none
	equality	== !=	none	== !=	none
Boolean	negation	not	left		
	conjunction	and	left		
	disjunction	or	left		
Sequential	prefix	->	right	->	right
	timed prefix			-*n*->	right
	guard	&	none		
	sequence	;	left	;	left
Choice	untimed timeout	[>	left	[>	left
	interrupt	/\	left	/\	left
	external choice	[]	left	[]	left
	internal choice	|~|	left		
	timed timeout			[*n* >	left
	timed interrupt			/ *n* \	left
Parallel	interface parallel	[| |]	none	[| |]	none
	interleave	| | |	left	| | |	left
Other	conditional	if then else	none	if then else	none
	local definition	let within	none		
	lambda term	\ @	none		

of the same operator, e.g., multiple choices '[]', several processes in parallel '[| |]', or a sequence ';' of processes. They are all in the form:

operator variable : *value-set* @ *process*

Each instance is created by substituting values for a given variable from the set, so the number of instances equals the size of the set. For example, this code would create the sequence `P(1); P(2); P(3)`:

```
;x:{1,2,3}@P(x)
```

Operators not implemented in CSP++ are either (a) not useful for synthesis, (b) can be rewritten in terms of other implemented operators, or (c) require support for datatypes and sets, which constitutes future work outside the scope of this thesis. Conversely, timed prefix, timeout, and interrupt are not available in FDR2. [Dox05] provides more details on implementation of CSPm operators in CSP++.

3.2.3 Untimed and Timed Timeout

The first operators we set out to implement were the untimed and timed timeouts. Let us recall the functionality of timeouts presented in Chapter 2. Semantics of the timeout operators resembles that of choice. Two versions of the timeout operator exist, however, the operator is most naturally timed. Consider this example,

$$P \overset{30}{\triangleright} Q$$

Process P only has to begin executing before 30 time units elapse in order to resolve the choice, i.e., only one of the two processes will get executed. As soon as P begins executing, process Q gets disregarded and cannot interrupt P.

In choice situations, CSP++ enforces the requirement of prefixing processes with an exposed first event. Consider, this choice situation written in standard CSPm:

```
CHOICE = P [] Q
P = a -> b -> SKIP
Q = c -> d -> SKIP
```

The first event, `a` or `b`, to occur in either `P` or `Q` will resolve the choice. CSP++, however, does not accept the above syntax, and the specification has to be rewritten like so:

```
CHOICE = a -> P′ [] c -> Q′
P′ = b -> SKIP
Q′ = d -> SKIP
```

Notice, the first events of `P` and `Q` were exposed and moved to `CHOICE`. This is to enable the translator to unambiguously identify the events which resolve the choice. Without explicitly exposing the first event, problems may arise with some definitions of `P` or `Q`. Consider, the example below:

```
CHOICE = P′′ [] Q′′
P′′ = a -> L [] b -> M [] c -> K
Q′′ = (d -> SKIP) ||| (e -> SKIP)
```

Now `P′′` has three potential first events, while `Q′′` has two. Far more complicated examples can be envisioned, in which it may be quite problematic to identify all of the candidate first events at translation time. The translation and execution of the choice operator are thus made tractable by simply forcing the specifier to write out the first events.

For the same reason, we expose the first event of the process on the left hand side of the timeout operator, like so:

```
a -> P [30> Q
```

In the above specification only the first event of process `P` must be exposed, but not `Q`. This is because process Q does not participate in a choice, i.e., there is no event of Q that we want to try to execute when making the choice. If event a was ready to be executed, CSP++ simply chains the execution to process Q, and we do not have to worry if Q is a complex combination of processes.

Table 3-3. Timed Timeout

CSP syntax	Generated C++ Code

`S = a -> REG [5> TIMEOUT`	`AGENTPROC( S_ )`
	`    Agent::startDChoice( 1 );`
	`        a();`
	`    if (Agent::whichTTChoice(5*timeunit)== 0){`
	`        CHAIN0( REG_ );`
	`    }`
	`    else {`
	`        CHAIN0( TIMEOUT_ );`
	`    }`
	`}`

Consider the example above in Table 3-3. We want to try executing event a for 5 time units before abandoning the left hand side of the timeout operator and continuing as process TIMEOUT. Now let us consider the generated C++ code in the right column of the table. `Agent::startDChoice(1)` method is borrowed from the implementation of the deterministic choice operator, with the exception that the CSP++ framework prepares to engage in a 1-way choice rather than n-way choice. Event a is tried once, and the `myChoice` member variable is set in the `Agent` class to the number of the successful event. In our case we only try for one event, so `myChoice` would be set to zero in case of a successful attempt at executing event a. `Agent::whichTTChoice()` will check the `myChoice` variable, and, if its value is zero, the function will return zero as well, making the `if` statement resolve the timeout in favour of the left hand side. However, if the initial attempt of `a` was unsuccessful—due to synchronization, for example—the `whichT-TChoice()` method will put the current thread (process `S`) to sleep for 5 time units. Variable `timeunit` contains the number of milliseconds in one time unit, and defaults to 1000 (=1 second) if not specified otherwise. (Refer to Appendix B for more information.) The thread wakes due to any of the following conditions:

- Event `a` succeeded, thus resolving the choice in favour of the left hand side of the timeout operator.
- 5 time units elapsed, and `a` event did not succeed, which would cause canceling further attempts of `a` and chaining to the `TIMEOUT` process.
- The process containing this timeout operator was interrupted due to being in the scope of an interrupt operator (see below for details). The immediate effect is to interrupt the left-hand side and abort the process.

Now let us focus on the untimed version of the timeout operator. As we already mentioned in Section 3.1.3, untimed timeout is by nature a non-deterministic operator. For software synthesis purposes, it did not make sense to implement non-determinism. Furthermore, there was no benefit in making the operator deterministic by copying the semantics of deterministic choice, as the latter operator is already implemented in CSP++. Thus, the decision was made to add determinism so as to provide extra functionality not available with '[]', but still within the formal definition of untimed timeout. As with the timed timeout, we require the first event of `P` to be exposed, like so:

```
a -> P [> Q
```

First, we try event `a`. If it succeeds, we continue as process `P` without ever considering process `Q`. However, if event `a` does not succeed, then we abandon the left hand side, and continue with process `Q`. The framework resolves the latter situation as "timed out."

The added functionality gives a designer the ability to check whether an event is ready without committing to keeping it on offer, as with the deterministic choice, and, if not, continue with an alternative. This resembles a polling action, where the timeout window does not depend on the first successful event of Q, but rather the attempt to execute the initial event of P.

Let us consider implementation details of the untimed timeout operator presented in Table 3-4.

Table 3-4. Untimed Timeout

CSPm Syntax	Generated C++ Code

`S = a -> REG [> TIMEOUT`	`AGENTPROC( S_ )`
	`    Agent::startDChoice( 1 );`
	`        a();`
	`    if (Agent::whichUTChoice()== 0){`
	`        CHAIN0( REG_ );`
	`    }`
	`    else {`
	`        CHAIN0( TIMEOUT_ );`
	`    }`
	`}`

Considering the example above, the idea for the untimed timeout operator is to poll event a, and see if the environment is ready to engage in it at the time of polling. Agent::startDChoice(1) prepares to engage in a 1-way choice with event a, just as it would for timed timeout. The Agent::whichUTChoice() method then checks the myChoice member variable, and, depending on its value, either returns zero signifying successful execution of the exposed event (a), or cancels further attempts to synchronize on the event. Unlike timed timeout's Agent::whichTTChoice(), the thread does not block. The if statement determines the subsequent flow of the program, either chaining to process REG on the left hand side of the timeout operator, or chaining to the TIME-OUT process.

3.2.4 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 after an event, before the next event can be engaged in. For example,

$$S = a \xrightarrow{3} b \rightarrow SKIP$$

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

CSPm syntax does not support the timed prefix operator, so we tried to make it resemble the prefix operator equipping it with an integer time value: `-n->`. Generated code for the above sample is shown in Table 3-5.

Table 3-5. Timed Prefix

CSP syntax	**Generated C++ Code**

`S = a -3-> b -> SKIP`	`AGENTPROC( S_ )`
	`   a();`
	`   Agent::nap(3*timeunit);`
	`   b();`
	`   END_AGENT;`
	`}`

In this example, process S will execute event a, then go to sleep for 3 units of time, before attempting to execute event b. Now let us consider the generated C++ code. As the execution begins, event a is performed, and we move on to the `Agent::nap()` method. It invokes the `task::timesleep()` method that blocks the current S process by putting it to sleep for 3 time units. Timed prefix's implementation only affects, or blocks, the CSP process (represented by a Pth thread) that includes timed prefix in its specification. Note that if included other processes, `Agent::nap()` would only block `S`'s execution, leaving all other threads to carry on. `Agent::nap()` is interruptible and may throw an exception, as described in the next section.

3.2.5 Untimed and Timed Interrupt

Let us recall the interrupt operator functionality. Interrupt implies concurrency, and is most naturally untimed. Consider this simple specification of the untimed interrupt,

$$P \,\Delta\, e \rightarrow Q$$

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. This is in contrast with the timeout operator, where P only has to start its execution in order to resolve the choice situation. With interrupt, process P must finish its execution before interrupt to avoid cancellation.

Interrupt operators proved to be the hardest to implement, partially due to the fact that CSP++ was never designed to be interrupted, based, as it is, on non-preemptive GNU Pth threads. This means that once a particular thread of execution gets control of the CPU, it will only release it if a blocking situation occurs, such as input/output or sleeping on a condition, or the thread will explicitly yield its control in a cooperative way. The solution was to make every blocking method in the framework interruptible, i.e., return control to the blocking method before the condition it is blocked on is fulfilled. For instance,

```
LEFT /\ inter -> RIGHT
```

By convention we start with the right hand side of interrupt operator, i.e., we test to see if the interrupting event can happen right away. We must be clear on what it means for the interrupting event to happen. For the interrupting event to happen, it must be recorded in the process's trace by successfully returning from the appropriate function call corresponding to every event. If the interrupt happens, then process `LEFT` is never even started. However, event `inter` may not successfully return from its function call and be blocked due to synchronization. This gives process `LEFT` a chance to run. Given the non-preemptive GNU Pth scheduler, how can the interrupting event `inter` happen when process `LEFT` is running? For interrupt to work, process `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. Process `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 execution trace after the interrupt. Every process in `LEFT`'s subtree must terminate and join with any threads that are waiting for it. Eventually, process `LEFT` will join with the threads it spawned and will also terminate. This is in contrast with the interrupt in the computer sense, where if a process is interrupted, the interrupt is serviced, and the scheduler goes right back to the interrupted process to continue its further execution.

Note, that 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, so now we were left with three possibilities:

1. spawn a new thread for process P, and let the already running process S attempt event e.
2. spawn a new thread that attempts event e, and let process S continue as process P.
3. spawn two threads: one to execute process P and one to attempt event e, while the already running process S waits for either of the two threads to complete.

The second possibility of spawning a new process to try event e was not realistic, as we would need to ensure the new spawned off process gets control of the CPU right away to see if the interrupt can be performed. This would require manually blocking the running process, starting a new thread that tries the interrupting event, and unblocking the thread running process P to either abandon it, if the interrupt succeeded, or let it run, if the interrupt did not succeed right away. The thread responsible for trying event e would have to be started with higher priority than the rest of the threads to make sure it gets control of the CPU to actually attempt e whenever the running P process blocks. This seemed like a very messy solution that is hard to understand and implement.

The third possibility of spawning two new threads, one for process P and one for event e, was more promising. However, we would be left with either an idle process S that simply waits for one of the two newly created threads to finish, or a coordinating process that regulates the execution of the two new threads, almost like a scheduler that gives control to the thread trying event e when thread P blocks, and switching back to P, if e is not ready. This simply led to excessive overhead of unnecessary switching.

In the end, the first choice of spawning a new thread for process P and letting S try event e featured the cleanest solution. It used the minimum number of threads avoiding unnecessary overhead, and was fairly simple to implement when compared to the second choice with messy code. The solution is described in detail later in the chapter.

The initial design for the interrupt operator was focused around Pth thread cancellation techniques. `pth_cancel(`*`tid`*`)` function call takes a thread id as a parameter and cancels further execution of the thread at one of its cancellation points (blocking calls, such as input/output). Unfortunately, after closer examination, it became apparent that such thread cancellation method will not suit our needs. For example, if a process `P` registered to participate in synchronization on an event `a`, but while waiting for the other participating process `Q` to complete the synchronization, `P` was cancelled. Using `pth_cancel()` method would not give the thread running process `P` a chance to unregister `a` from synchronization. When `Q` would finally try to complete synchronization on event `a`, CSP++ would fail issuing a `segfault` error because it would not be able to find the thread running `P`. Therefore, Pth thread cancellation utilities were not useful for our purposes.

Further research yielded another solution to the interrupt problem. This time the focus shifted from the interrupting process to the interrupted one to do its own clean-up and then self-terminate. Recall that only blocked threads, i.e., threads that released control of the CPU, can be interrupted. When a thread is under the scope of the interrupt operator and enters a blocking function call, it needs to be explicitly woken up/alerted before the condition it is waiting for is fulfilled. Once the interrupted thread is woken up, it checks to see the reason for its wake-up, whether the blocking condition was fulfilled or an interrupt occurred. If it is indeed interrupted, the thread performs its own clean up, deallocating memory, backing out of partially-synchronized events, and so on, at which point it needs to short-circuit back to the `Agent::run()` method. `Agent::run()` controls the execution of every `Agent` and is the place where further execution can be abandoned. The C++ exception mechanism proved to be effective for redirecting the control flow of the interrupted thread. Once the interrupted thread finishes cleaning up, it throws an exception that is caught in `Agent::run()`, and the thread's further execution is abandoned. This was important because within the "`AGENTPROC`" function body generated for each CSPm process, individual event executions—which are, in fact, invocations of methods—have no status return to check whereby they could abort the function via, say, returning prematurely. To add such apparatus would be a major change to the existing code generation methodology. In contrast, throwing a C++ exception can be done from inside an invoked method, and is a clean way to abort the method caller's function. This also takes care of unwinding the stack and invoking any associated destructors, thereby automatically freeing the dynamic storage that is under control of stack variables such as `FreeVar`.

Let us review the particular points at which any given thread can be interrupted: The only requirement for a thread to be interrupted is that it must block, releasing control of the CPU. Table 3-6 lists all the functions in the CSP++ framework which can be interrupted.

There were other challenges encountered during interrupt implementation. The problem of getting CPU control to an interrupting event has already been raised. When an interrupt occurs, and the process under the scope of the interrupt operator has entered a blocking function call, we need to make sure the parent process—which took over the right side of the interrupt operator — gets control of the CPU during the first context switch in order to finish executing the interrupting event. Maximizing the thread priority for the parent process increases its chances of being executed next. We cannot *guarantee* which thread is going to be dispatched on the next context switch, since this is done under the Pth scheduler's control, however, raising the priority of the parent thread is good enough because we only want to "beat" the priority of the process which has to be interrupted, the

Table 3-6. Interruptible Methods in CSP++

Method Name	Description	Necessary clean-up upon interrupt
`Agent::whichDChoice()`	In deterministic choice situation, once all exposed events are tried initially and none succeed, this method attempts to re-execute the events, only to wake up when one happens	Cancel further attempts of re-executing deterministic choice events.
`Agent::whichTTChoice()`	In timed timeout situation, left hand side of the operator has a certain amount of time to engage in an external event, during which time the thread is put to sleep	Cancel further attempts at re-executing the left hand side exposed event.
`Agent::nap()`	Time delay between two successive events puts the current thread to sleep.	Abandon execution of the current routine, no clean up.
`Agent::doSync()`	Event synchronization.	If waiting for synchronization flag, abandon waiting. No further clean up. If waiting for synchronization, return `SR_INTERRUPTED` status to caller, `Action::execute()` and `Action::reexecute()`, which unsynchronize further attempts at event synchronization.
`Agent::startUI()`	Untimed interrupt. Sleeping while waiting for a wake up call due to left hand side completing or interrupt event happening.	Cancel further attempts at executing the interrupt event.
`Agent::startTI()`	Timed interrupt. Sleeping while waiting for a wake up call due to left hand side completing or time elapsing.	Abandon execution of the current method.

one on the left hand side of the interrupt operator. Since the left hand process starts up with a default priority of zero, raising its parent's priority to the maximum of 5 will always win, thus ensuring the parent's ability to run once the child blocks. After the parent process finishes its handling of the interrupt — or when the left hand side completes without having been interrupted — its priority is returned to the default level since it no longer needs special treatment.

It was also necessary to prevent an interrupt from happening once the left hand process completed its execution, i.e., when there would be nothing to interrupt. For instance,

```
S = LEFT /\ inter -> RIGHT
```

If process `LEFT` completes its execution before event `inter` happens, then all attempts at executing the interrupt should be canceled. On the other hand, if the `inter` event happens before `LEFT` completes its execution, further progress of `LEFT` must be canceled. Hence, process `S` has to be alerted when one of the two things described above happened. The mechanism for alerting `S` when the `inter` event happens was already implemented in the framework, but we still needed to add a mechanism for alerting the parent thread (`S`) when its child (`LEFT`) completed its execution. Every CSP process is implemented as an object of the `Agent` class, which in turn is a subclass of `task` subclassed from `object`, which has a list of waiters associated with it. Every parent process would put itself on its child's waiters list to be alerted when the child completes its execution.

A way had to be devised to manage a nested interrupt environment. For example,

```
S = (LEFT /\ inter -> RIGHT) /\ inter_top -> TOP
```

`LEFT`'s execution can be interrupted by either `inter` or `inter_top`, while `RIGHT` can only be interrupted by `inter_top`. A new `EnvInt` class that inherits from both `Env` class and `object` (so it can be alerted/waited on) was defined. It points up the `Agent`'s environment stack to its parent `EnvInt`, if any. A `bool interrupted()` method was added to `EnvInt` class, which returns true or false depending on whether an interrupt took place. The `Agent` class was outfitted with a new data member that points to an `EnvInt` object just pushed on the environment stack, or has `NULL` value if it is not subject to interrupt. When a new `Agent` object is created, it automatically copies this data member from its parent `Agent`, so as to replicate the enclosing interrupt environment, while avoiding the need for a stack search. In addition, if the `Agent` is subject to an interrupt operator, it "remembers" itself on all applicable `EnvInt` objects' waiters lists. When an `Agent` wakes up from sleep in a blocking function, it calls the `isInterrupted()` method to find out whether or not its execution was canceled. If the `Agent` finds out it was interrupted, it performs its clean up procedures described in Table 3-6, and throws an exception that is caught in the `Agent::run()` method, which controls execution of every `Agent`. When the exception is caught, `Agent::run()` stops further execution of the current thread. This satisfies the CSP requirement for the interrupt operator that no more events of the interrupted process appear in the execution trace.

Let us consider an untimed interrupt example in Table 3-7. First, looking at the left column, when execution of process `INT` begins, event `inter` is tried. If `inter` succeeds right away (which it does not in this case due to synchronization with `SYNCER` process, which has not started yet), process `LEFT` does not even get a chance to run, and process `RIGHT` follows the execution of `inter`. In this case `inter` does not happen at first, and process `LEFT` gets a chance to execute, but it may be interrupted by `inter` which will cause abandoning of the execution of `LEFT` and its subtree (interleaved combination of processes `LEFT1` and `LEFT2`) in favour of process `RIGHT`. `LEFT1` and `LEFT2` have blocking delays of two time units represented by timed prefixes. Timed prefix is implemented by the `Agent::nap()` method, one of the interruptible functions listed in Table 3-6 and is the place were processes `LEFT1` and `LEFT2` could potentially be interrupted. Process `SYNCER` is used to synchronize on event `inter`, effectively delaying its execution in process `INT` until process `SYNCER` gets a chance to run. Without `SYNCER`,

Table 3-7. Untimed Interrupt

CSPm syntax	**Generated C++ Code**
`INT = LEFT /\ inter -> RIGHT`	`AGENTPROC( INT_ )`
	`   int choice_;`
`RIGHT = did_right -> SKIP`	`   Agent::startDChoice( 1 );`
	`      inter();`
	`   if ( Agent::whichUIChoice() == 0) {`
`LEFT = LEFT1 ||| LEFT2`	`      CHAIN0( RIGHT_ );`
`LEFT1 = b -2-> c -> SKIP`	`   }`
`LEFT2 = d -2-> e -> SKIP`	`   else {`
	`      inter_r.interrupt();`
`SYNCER = a -> inter -> SKIP`	`      CHANGEPRIO(5);`
	`      Agent* a1 = START0( LEFT_, 0 );`
`SYS = INT [|{|inter|}|] SYNCER`	`      choice_ = Agent::startUI(a1);`
	`      CHANGEPRIO(0);`
	`      WAIT( a1);`
	`      Agent::popEnv(1);`
	`      if(choice_ == 1) {`
	`         CHAIN0( RIGHT_ );`
	`      }`
	`   }`
	`   END_AGENT;`
	`}`

the framework will engage in event `inter` without ever starting process `LEFT`. However, because process `INT` is the first to run (in CSP++, leftmost processes are always started first), its offer to synchronize on `inter` cannot be fulfilled as `SYNCER` has not had a chance to run yet, thus execution continues with process `LEFT`.

When the interrupt operator '/\' is encountered in the CSP specification, the C++ code in the right column of Table 3-7 is generated. When execution begins, `Agent::startDChoice(1)` attempts to execute `inter` event to see if the interrupt can happen immediately, but fails due to inability to synchronize with `SYNCER` at this point. `Agent::whichUIChoice()` sees that the initial attempt at `inter` failed, but synchronizing on `inter` remains on offer. `Agent::whichUIChoice()` returns with the appropriate status causing the program to enter the `else` clause. The `interrupt()` method creates a new `EnvInt` object and pushes it on the `Agent`'s environment stack. `INT`'s priority is raised to ensure it gets control on the next context switch. Since, the initial attempt at `inter` failed, process `LEFT` is started by `Agent* a1 = START0( LEFT_, 0 )` with a default thread priority of zero. At this point process `LEFT` is put in the ready queue of the Pth scheduler to be dispatched after the next context switch. `Agent::startUI(a1)` receives a pointer to the newly created thread for process `LEFT`. The function (1) puts `LEFT` onto `EnvInt` object's waiters list so that `LEFT` can be alerted in case of interrupt, (2) "remembers" the `INT` process on `LEFT`'s waiters list so that process `INT` can be informed if `LEFT` finishes its execution before the interrupt (making `INT` cancel further interrupt attempts), and (3) puts the current thread to sleep to be woken up either due to `LEFT` completing its execution or interrupt happening. Now, process `LEFT` gets control of the CPU and starts the interleaved execution of `LEFT1` and `LEFT2`, both of which encounter blocking functions when executing delays of two time units. Processes `LEFT1` and `LEFT2` lose control of the CPU. The Pth scheduler performs a context switch and gives `SYNCER` a chance to run. At this point `SYNCER` completes synchronization with `INT` on event `inter`, as `inter` has been "on offer" since first being tried. After `SYNCER` completes its execution, CPU control is returned to `INT` as synchronization on `inter` is completed and a wake up call to `INT` is issued. When woken up in `startUI()` method, `INT` sets `EnvInt` object's `interrupt` data member to true, and alerts `LEFT1` and `LEFT2` to wake up giving up its control of the CPU. Upon wake-up, `LEFT1` and `LEFT2` call `EnvInt`’s `isInterrupted()` method to find out that both of them were in fact interrupted while sleeping in `Agent::nap()` methods. `LEFT1` and `LEFT2` `Agent::nap()` methods perform clean-up (see Table 3-6) and throw exceptions that return control to `Agent::run()` method of their respective processes, at which point their further execution is aborted. Control is once again returned to process `INT`. `Agent::startUI()` method returns, setting `choice_` variable to 1. At this point `INT` process does not need special access to CPU control anymore, so we lower its priority back to the default value of zero. The `WAIT(a1)` macro joins the current `INT` thread with the aborted `LEFT` thread. Finally, `INT`’s execution is chained to process `RIGHT`.

When executing the above specification, CSP++ produces the trace <`b, d, a, inter, did_right`>, satisfying the requirements for the interrupt operator.

Implementation of the timed interrupt follows the same logic as the untimed interrupt with one exception: an interrupting event is no longer needed, as the interrupt happens due to time elapsing.

It is no longer necessary to execute/re-execute the interrupting event, so `startDChoice()` and `whichUIChoice()` methods are not generated. If interrupt time was specified to be zero in the specification, then we chain to the interrupting process right away, never even considering process `LEFT`. Otherwise we follow the same logic as with the untimed interrupt, except that the `EnvInt` object is set to “interrupted” when the specified time elapses. Nevertheless, let us examine the above timed interrupt example in more detail.

First, consider the CSP specification in the left column of Table 3-8. Execution begins with process `S`. Process `S` gives process `LEFT` two time units to execute. If two time

Table 3-8. Timed Interrupt

CSP syntax	Generated C++ Code
S = LEFT /2\ RIGHT	AGENTPROC(S_)
	int time = 2*timeunit;
RIGHT = did_right -> SKIP	if (! time) {
	CHAIN0(RIGHT_);
	}
LEFT = LEFT1 ||| LEFT2	else {
LEFT1 = a -4-> b -> SKIP	inter_r.interrupt();
LEFT2 = c -4-> d -> SKIP	CHANGEPRIO(5);
	Agent* a1 = START0(LEFT_, 0);
	choice_ = Agent::startTI(a1, time);
	CHANGEPRIO(0);
	WAIT(a1);
	Agent::popEnv(1);
	if(choice_ == 1) {
	CHAIN0(RIGHT_);
	}
	}
	END_AGENT;
	}

units is enough for LEFT to finish its execution and terminate, process RIGHT will never happen. However, if LEFT does not finish its routines in two time units, process RIGHT will interrupt LEFT. Given two time units to execute, LEFT starts its execution by kicking off interleaved execution of LEFT1 and LEFT2. LEFT1 performs event a and blocks for four time units, while LEFT2 performs event c and also blocks for four time units. While LEFT1 and LEFT2 remain block, two time units allocated for LEFT's execution elapse and process RIGHT cancels further execution of LEFT1 and LEFT2, performs did_right and successfully terminates.

Now, let us examine how CSP++ handles the above specification illustrated by C++

code in the right column of Table 3-8. When execution of AGENTPROC(S_) begins, interrupt time is set to the appropriate time units. If the interrupt time was specified as zero, process RIGHT would have taken over right away without ever giving LEFT a chance. In our case, the interrupting time is two time units, so we enter the else clause. The interrupt() method creates a new EnvInt object and pushes it on the Agent's environment stack. S's priority is raised to ensure it gets control on the next context switch. Process LEFT is started by Agent* a1 = START0(LEFT_, 0) with a default thread priority of zero. At this point process LEFT is put in the ready queue of the Pth scheduler to be dispatched after the next context switch. Agent::startTI(a1, time) receives a pointer to the newly created thread for process LEFT and time allocated for LEFT's execution. The function (1) puts LEFT onto EnvInt object's waiters list so that LEFT can be alerted in case of interrupt, (2) "remembers" the S process on LEFT's waiters list so that process S can be informed if LEFT finishes its execution before the allocated time (making S cancel further interrupt attempts), and (3) puts the current thread to sleep to be woken up either due to LEFT completing its execution or two time units elapsing. Now, process LEFT gets control of the CPU and starts interleaved execution of LEFT1 and LEFT2, both of which encounter blocking functions when executing delays of four time units. Processes LEFT1 and LEFT2 block, and lose control of the CPU. At this point two time units elapse, and the Pth scheduler wakes up S in startTI() method. When woken up, S sets EnvInt object's interrupt data member to true, and alerts LEFT1 and LEFT2 to wake up giving up its control of the CPU. Upon wake-up, LEFT1 and LEFT2 call EnvInt's isInterrupted() method to find out that both of them were in fact interrupted while sleeping in Agent::nap() methods. LEFT1 and LEFT2

`Agent::nap()` methods perform clean-up (see Table 3-6) and throw exceptions that return control to `Agent::run()` method of their respective processes, at which point their further execution is aborted. Control is once again returned to process `S`. `Agent::startTI()` method returns, setting `choice_` variable to 1. At this point `S` process does not need special access to CPU control anymore, so we lower its priority back to the default value of zero. The `WAIT(a1)` macro joins the current `S` thread with the aborted `LEFT` thread. Finally, `S`'s execution is chained to process `RIGHT`.

The above example will produce the following trace <`a, c, did_right`>, satisfying the requirement for the timed interrupt.

3.3 Implementation restrictions and limitations

The following list presents current limitations of Timed CSP++:

- CSPm syntax only supports 2 of the 5 operators we set out to implement: untimed timeout and untimed interrupt. To continue using FDR2 and ProBE, specifications written in Timed CSP would have to be lexically trimmed down, leaving out any timing information. Timed interrupt, timed timeout, and timed prefix would have to be changed to their untimed counterparts, which is easy to do with a simple script. To fully explore the correctness of Timed CSP specifications one would have to use the HORAE tool, which, unfortunately, is not widely available at the moment.
- Compared to Timed CSP, the timeout operators in CSP++ force the user to expose the first event of the left hand side process. This is compatible with CSP++'s treatment of the 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 side process, performing internal events, but rather begins its execution once the exposed interrupting event happens.

User-coded functions do not have any special mechanism for blocking, beyond using Pth-wrapped system calls that block only the calling thread rather than the entire Unix process. This means that UCFs are not well-integrated with interrupts. 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 weakness of CSP++’s interface to UCFs, which warrants further study. Changes to that interface were beyond the scope of this thesis.

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.

4 VAC Automated Cleaner case study

The Disk Server Subsystem (DSS) [Gar00] and the Automated Teller Machine (ATM) [Dox05] CSP++ case studies illustrated implementation of many CSP features as well as the selective formalism approach incorporating UCFs. However, these case studies could not feature any timing constructs. In this chapter, we will demonstrate the use of the new CSP++ timing constructs with a new VAC Automated Cleaner (VAC) case study.

Unfortunately, formal verification and state space exploration tools such as FDR2 and ProBE could not be used to verify the correctness of the specified VAC system as they do not support Timed CSP. VAC can be stripped of any timing information, of course, and FDR2 and ProBE can be applied. However, as we have seen from examples, sometimes the added timing information can completely 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. Therefore, it was important to find a way to formally verify Timed CSP specifications. Fortunately, while undertaking this research we came across such a tool. A team at the National University of Singapore School of Computing has been working on HORAE [DZSH06], a tool similar to FDR2 used to reason about Timed CSP specifications. The tool has not yet been released, so we can only show how one can potentially verify Timed CSP specifications using HORAE in the future.

This chapter starts by describing the VAC case study, and then turns to formal verification. Since HORAE is not available to verify VAC, we can at least show how the new timed operators in CSP++ make it possible to synthesize code for the HORAE examples published so far. Finally, we discuss limitations of CSP++'s current interface to UCFs in regard to fully exploiting the new functionality of timed operators, with suggestions for future work.

4.1 VAC Design

VAC implements a simple robot-vacuum, which, if left in automatic mode, drives around a room picking up dust and avoiding obstacles, but can also be controlled manually through a remote. As was done with the ATM case study [Dox05], the VAC design can be divided into 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 can be attributed to the functional model. Environmental entities providing stimuli to VAC, such as a user and room layout can be attributed to the environmental model. The environmental model is very useful during system design and testing stages, when providing physical input to the system may 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. Figure 4-1 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.

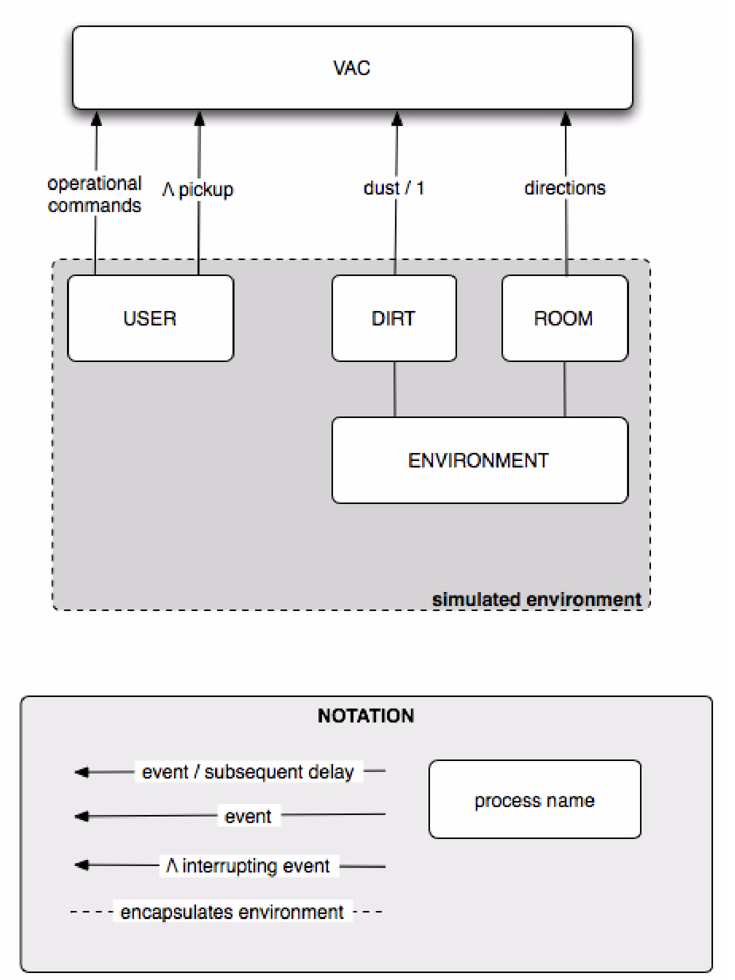

Figure 4-1. VAC Interaction with the Environment

Consider the functional model. Instead of focusing on complex robot AI 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 timed CSP primitives contribute to the overall behaviour of the specified system. VAC specification includes all new CSP operators added to CSP++ as part of this research. In the following sections we will give detailed description of VAC and its parts, and illustrate integration of the Timed CSP operators. We will also identify the events that can be linked to user-coded Functions, which would perform physical work if the robot were to be implemented. First, let us consider Figure 4-2 which visually portrays the overall design of VAC using State-Charts. The visual representation of VAC gathers all interacting components in one place and helps grasp the final goal before it is written in CSPm. The full CSPm specification is listed in Appendix D.

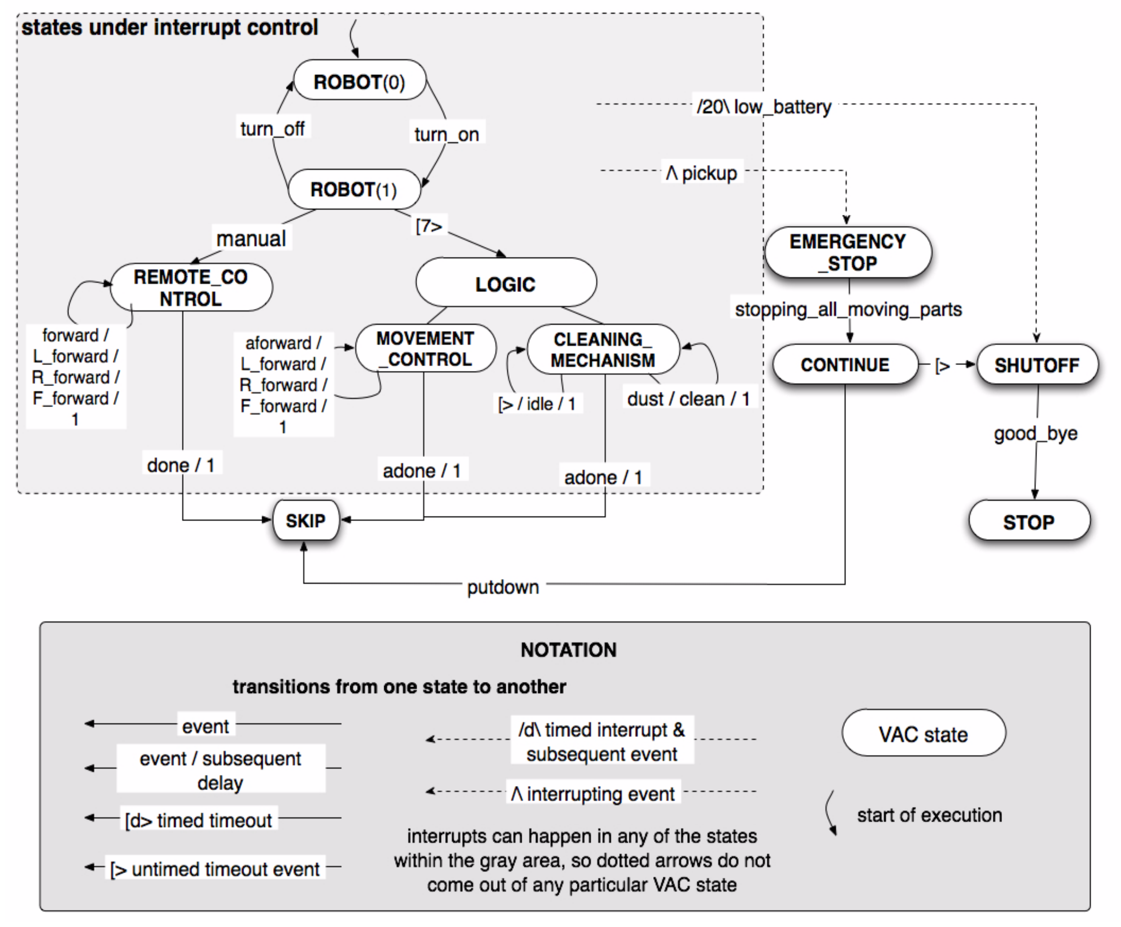

Figure 4-2. VAC Statechart

The following statement says that the execution begins with the parallel execution of the functional model (`ROBOT(0)`) with one environmental model (`USER`), and explicitly states all the events which the two models must synchronize on.

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

Initially VAC is turned off and is only ready to respond to the `turn_on` event, represented by `ROBOT(0) = turn_on -> ROBOT(1)`. Once turned on and running, the VAC can only operate for 20 time units—simulating a limited battery life—and at any point can be picked up by the user causing an interrupt and suspension of all moving parts:

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

VAC features two modes of operation: automatic, described by process `AUTOMATIC_MODE`, and manual, described by process `REMOTE_CONTROL`:

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

To trigger one or the other, the user can either select `manual`, which will cause the VAC to accept further commands from the user in `REMOTE_CONTROL`, or leave the VAC to wait 7 seconds for a possible `turn_off`, and then go into automatic mode.

The latter is controlled by `LOGIC` and synchronizes with simulated environment (`ROOM` and `DIRT`):

```
AUTOMATIC_MODE  = ENVIRONMENT
     [|{|aforward, abackward, aleft, aright, adone,
     astop, dust|}|] LOGIC
ENVIRONMENT = ROOM ||| DIRT
ROOM = aforward -1-> aleft -1-> aforward -1->
     aright -1-> abackward -1-> adone -> SKIP
DIRT = dust -1-> DIRT
```

`ROOM` and `DIRT` are running as independent interleaved processes, simulating obstacles and dust particles in the room, respectively. These processes act as constraints on the choices in `LOGIC` via synchronizing events.

The `LOGIC` process controls the interleaved operation of `MOVEMENT_CONTROL`, responsible for driving the robot around the room by synchronizing with directions provided from the environment (`ROOM`), and `CLEANING_MECHANISM`, which cleans dust by synchronizing with the environmental process `DIRT` on event `dust`, or remains idle until the `done` command is received, simulating successful termination of room cleaning:

```
LOGIC = MOVEMENT_CONTROL
          [|{|adone|}|] CLEANING_MECHANISM
MOVEMENT_CONTROL =
     (aforward -> L_forward -> R_forward ->
      F_forward -1-> MOVEMENT_CONTROL) []
     (abackward -> L_backward -> R_backward ->
      F_backward -1-> MOVEMENT_CONTROL) []
     (aleft -> L_backward -> R_forward ->
      F_turn -1-> MOVEMENT_CONTROL) []
     (aright -> L_forward -> R_backward ->
      F_turn -1-> MOVEMENT_CONTROL) []
     (astop -> L_stop -> R_stop -> F_stop ->
      MOVEMENT_CONTROL) []
     (adone -1-> SKIP)
CLEANING_MECHANISM = (adone -1-> SKIP) [>
     ((dust -> clean -1-> CLEANING_MECHANISM) [>
      (idle -1-> CLEANING_MECHANISM))
```

VAC execution can be terminated in three ways:

- successful termination,
- user picking up the robot, triggering untimed interrupt, or
- low battery, triggered by the passage of time.

Now, let us consider what events could be linked to user-coded Functions to provide more functionality to the VAC. Appendix D provides, along with the full CSPm specification of VAC, the generated C++ code and the ucfs.cc file, which includes functionality for the abstract events listed in Table 4-1. The presented UCFs are only meant to provide proof of concept for demonstration. When the generated C++ code is run and UCFs are triggered, they only output strings to the console, which are meant to represent hardware functionality of VAC.

Table 4-1. User-coded Functions in VAC

Specification event	User-coded Functionality
`turn_on, turn_off`	Interface to the POWER button on the VAC
`pickup, put_down`	Interface to touch sensors which determine whether the VAC is on the ground or not
`L_forward, L_backward, L_stop, R_forward, R_backward, R_stop, F_forward, F_backaward, F_turn, F_stop`	Actual movement of the appropriate wheel in the correct direction
`low_battery`	Interface to the VAC's battery
`clean`	Rotation of the cleaning drum and air suction control
`stopping_all_moving_parts`	Mechanism that stops the rotation of the wheels, cleaning drum, and air suction

The use of each of the new timed operators within VAC will now be highlighted in the following subsections.

4.1.1 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 the timed prefix operator.

In MOVEMENT_CONTROL on page 80, 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 one-unit delay suspends the process's execution and puts it to sleep, only to be woken up either after (at least) one time unit has elapsed or due to an interrupt. It should be noted, that all time units represent minimum delays due to thread non-preemption, that is, the thread will sleep, at a minimum, one time unit (if not interrupted), but may sleep longer. The delays leading to thread suspension give interrupts a chance to execute. Without delays, threads controlling interrupts would never get a chance to execute, leading to difficulty in modeling interruptions in VAC's execution.

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.

4.1.2 Timeouts

VAC illustrates the use of timeout operators in two cases:

```
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 `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 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. 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. There are no additional attempts to execute the left hand side event. This polling operation adds a new programming tool to CSP++, not available before.

4.1.3 Interrupts

Interrupt operators are the most complex operators added to CSP++. VAC illustrates the use of untimed and timed interrupts. Interrupts force already-running processes to abandon their further execution from the point of the interrupt, while further flow of control is defined by the interrupting process.

Untimed interrupt is illustrated in the following example:

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

Say, the user turned the robot on and left it. After some time, when no more commands are issued from the user, the robot will timeout in favour of automatic execution. 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, picking the working robot up at any point is the perfect scenario 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.

4.1.4 VAC testing

The VAC case study was translated, compiled, and executed with a number of different variations in the environment to mimic different execution paths in the attempt to discover any deadlock situations. Unfortunately, formal verification of VAC is not possible at present, as HORAE, the timed formal verification tool (described in the next section), was not available for testing. Appendix D includes the complete CSP specification of VAC with several environmental models available for immediate manual testing.

4.2 CSP++ and Formal Verification

Currently, there are two ways of verifying specifications written in CSP. FDR2 and ProBE fully support the untimed CSP language, and have proven themselves as industry standards when it comes to formal verification of CSP specifications. During this research CSP++ was extended with several new CSP primitives, two of which are untimed timeout and untimed interrupt. The two operators are supported by FDR2 and ProBE. Hence, CSP specifications that include these operators can be verified by the commercial tools.

HORAE, a new tool being developed at the National University of Singapore, supports Timed CSP, however, it is still under development and has not been released. It uses Constraint Logic Programming (CLP) to reason about Timed CSP. [DZSH06] references some of the case studies and advantages, such as expressiveness, of using CLP as the underlying reasoning logic.

HORAE, shown in Figure 4-3, 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" [DZSH06] and is used to verify variable bound properties. Both modules are responsible for checking the safety and liveness of a given Timed CSP specification.

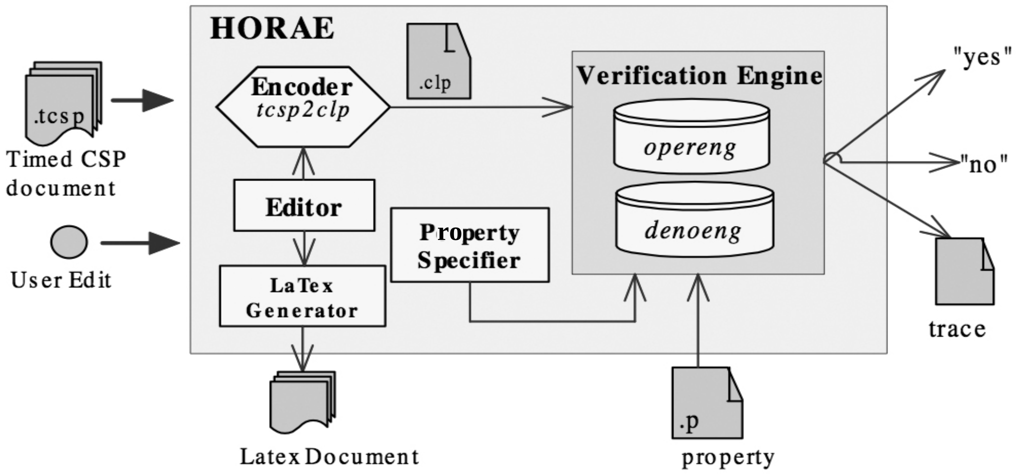

Figure 4-3. HORAE Design (from [DZSH06])

The front-end of HORAE includes a text editor used to build Timed CSP specifications in machine readable Timed CSP format represented by ASCII characters. HORAE then converts Timed CSP specification files (.tcsp) into CLP syntax suitable for verification with CLP(R). Table 4.1 presents HORAE syntax for Timed CSP operators (the full list of HORAE supported operators can be found in Appendix C). The symbol *d* represents an integer constant holding a time interval value.

HORAE syntax for timed operators resembles function calls, while we tried to make our syntax look like CSPm syntax 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 substantial, it is fairly easy to make specifica-

Operator	**CSP syntax**	**HORAE .tcsp syntax**	**CSP++ syntax**
Delay	$a \xrightarrow{d} b$	->{d}	a -d-> b
Untimed Timeout	$Q1 \triangleright Q2$	not supported	Q1 [> Q2
Timed Timeout	$Q1 \overset{d}{\triangleright} Q2$	|\\{d}	Q1 [d> Q2
Untimed Event Interrupt	$Q1 \Delta a \rightarrow Q2$	int(Q1, a, Q2)	Q1 /\ a -> Q2
Timed Interrupt	$Q1 \Delta_d Q2$	tint(Q1, d, Q2)	Q1 /d\ Q2

Table 4-2. Timed CSP operators supported by HORAE and CSP++

tions 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. However, a general guide of how one might use CSP++ and HORAE will consist of the following steps:

1. Write a Timed CSP specification
2. Run CSPtoHORAE script to convert the specification syntax accordingly
3. Run the converted specification through HORAE
4. Adjust the specification in case HORAE detects deadlocks, livelocks, or other race hazards
5. Repeat steps 3 and 4 until the specification is free of any undesired behavior
6. (Optional) The specification may be stripped of any timing information and run through ProBE and FDR2 to gain additional information about system behaviour

7. Generate C++ code from the specification by using CSP++

8. (Optional) Depending on the purpose of the generated system, it may be extended with UCFs to perform specific tasks

Through e-mail correspondence with the HORAE team we were able to obtain Timed CSP specification files that have been used to test the HORAE tool. These specifications described small, popular case studies:

- **Dining Philosophers**. A classic untimed deadlock example featuring *N* philosophers and *N* forks. All philosophers sit at a round table with forks placed between two neighboring philosophers. Each philosopher needs two forks to eat.
- **Timed Vending Machine**. After inserting a coin, a user has 10 seconds to make a choice between coffee, tea, or request to release the coin. If the user does not make his choice in the allocated time, the vending machine times out and releases the inserted coin.
- **Timed Railroad Crossing**. The system is composed of three components: a train, a gate, and a controller. The gate is up and allows traffic to pass through the crossing if there is no train. The gate goes down if a train is approaching. The controller monitors the approach of trains and signals the gate to be lowered before the train enters the crossing. Timing information was specified as follows: the train needs 5 minutes to reach the crossing from the moment the controller spots it and 20 seconds to pass through the crossing; the controller needs 1 second from the moment a signal from train approaching or leaving is received to the moment it relays the appropriate message to the gate; the gate needs 100 seconds to raise or lower the bar.

After adjusting the syntax to fit CSP++ we ran these specifications through our system to get executable versions. Running executable versions of the three case studies produces execution traces, an ordered list of executed events. This gave us a chance to do some comparison with HORAE's claims described in [DHSZ06]:

Dining Philosophers. Two variations of the problem were examined: 3 philosophers and forks, and 4 philosophers and forks. HORAE was able to verify the following properties: (1) both variations are not deadlock-free; (2) "No more than N+1/2 philosophers can eat at the same time." The authors probably overlooked a mistake in the formula, as it is unclear what they meant: $\frac{N+1}{2}$ or $N+\frac{1}{2}$. However, both formulas are not correct, because when *N*=3, only 1 philosopher can obtain two forks and eat, while when *N*=4, 2 philosophers can eat at the same time leading to $\frac{N-1}{2}$ for all odd *N* and $\frac{N}{2}$ for all even *N*; (3) It is possible to have one philosopher eat all the time with the others starving. This case only applies to *N*=3 problem, but not *N*=4. Unfortunately, the authors did not specify this detail. The authors were also able to confirm the first and the third property with FDR2. We were only able to confirm the deadlock property with FDR2. When running the *N*=3 example through CSP++, the following trace was produced: `<enter1, enter2, enter3, pick11, pick22, pick33>` leading to deadlock.

Timed Vending Machine. The following properties were verified: (1) the specification is deadlock-free; (2) trace timewise refinement properties, i.e., whether timed execution traces were valid traces of the system; (3) whether there is a case such that coffee is selected while tea is dispatched (assuming 'no', as the authors do not state the answer explicitly). The authors also verified trace refinement properties in FDR2 dropping the timed timeout operator and replacing it with deterministic choice.

Running this case study through CSP++ required the addition of one extra event not present in the original specification. Consider the following HORAE code:

```
TVM = insert ->
     ((reqrelease -> release ->{3} TVM) []
      (coffee ->{3} dispatchcoffee -> TVM) []
      (tea ->{2} dispatchtea -> TVM)) |\{10}
     (release -> TVM)
```

`reqrelease`, `coffee`, and `tea` are tried for up to 10 seconds to see which one succeeds and determines the subsequent flow of control. Due to the three-way choice, the scope of the timed timeout operator covers four possibilities. In contrast, the scope of the timed timeout operator in CSP++ covers only two possible outcomes: left hand side of the operator succeeds (only one exposed event can be tested with the timeout), or timeout occurs.

To overcome this, an extra event had to be inserted into the CSP++ specification:

```
TVM = insert ->
     (choice ->
          ((reqrelease -> release ->{3} TVM) []
           (coffee ->{3} dispatchcoffee -> TVM) []
           (tea ->{2} dispatchtea -> TVM))) |\{10}
     (release -> TVM))
```

This is as if pressing, say, ‘coffee’ caused two events: `choice` and `coffee`. This is a reasonable modification that does not materially change the semantics of the specifcation. Another approach is to substitute channel input, e.g., `button?n`, where `n`=1 for ‘release’, 2 for ‘coffee’, etc., in `(button?x -> ...) [> (release -> TVM)`. When run through CSP++, the test case was deadlock free producing valid traces of the system. This showed that results obtained with CSP++ are compatible with those of HORAE. Expanding the scope of the timeout operator is postponed for future work and may be address in subsequent releases of CSP++.

Timed Railroad Crossing. The following properties were verified: (1) the specification is deadlock-free; (2) `<trainnear, nearind, downcomm, down, confirm, entercrossing, leavecrossing, outind>` is a legal trace of the system; (3) the lowest time required for a train to pass through the crossing is 320 seconds. The first two properties were also confirmed by FDR2, assuming that to verify the second property, the authors removed all the timing information. When running this example through CSP++, the following trace was obtained: `<trainnear, nearind, downcomm, down, confirm, entercrossing, leavecrossing, outind, upcomm, up, confirm>` which satisfies the safety requirement for the gate to be down when a train is passing through the crossing.

The ability to execute these files via CSP++ shows that HORAE and CSP++ are very much compatible, and hopefully we will be able to verify more complex examples in HORAE in the near future. Full specifications of the three CSP case studies are presented in Appendix A.

4.3 Timed Operators and UCFs

The strongest feature of CSP++ that sets it apart from other CSP-inspired projects is the ability to decide exactly what part of a specified system needs to be formally specified, and what part can be plugged in later as user-coded functions. The addition of timing operators did not change the way UCFs can be used with CSP++, however, at the moment, there are some limitations that programmers should be aware of when using UCFs and timing operators together.

UCFs can happily co-exist with the timed prefix operator without any restrictions or limitations, however, they do not yield to the semantics of timeouts and interrupts. Consider the following Timed CSP example:

$$\text{ATM} = \text{read\_PIN} \rightarrow \text{OPTIONS} \overset{30}{\triangleright} \text{return\_card} \rightarrow \text{SKIP}$$

A user has 30 seconds to enter his PIN and receive other options, or the ATM will return the card. If we decided to design an actual ATM using CSP++, it would be reasonable to link read_PIN event to a UCF that actually reads the input from the ATM's numeric pad. Recall that performance of the first external event determines the flow of control in the timeout situation. In the ATM scenario, execution of read_PIN within 30 seconds would prevent the timeout from happening. Let us consider what "executing" an event means in CSP++. read_PIN would have to return from the UCF before 30 seconds to avoid the timeout. If read_PIN blocked waiting for the user's input, and did not return within 30 seconds, the desired flow of control would cause the timeout to happen and abort the UCF. However, there is currently no mechanism in the framework to get control back from the UCF.

Now let us consider a fragment of the VAC specification:

```
MOVEMENT_CONTROL = (aforward -> L_forward ->
     R_forward -> F_forward -1-> MOVEMENT_CONTROL) []
     ...
     (adone -1-> SKIP)
```

`L_forward`, `R_foward` and `F_forward` are good candidates for UCFs as they are responsible for rotating the appropriate wheels in the desired direction. Recall that `MOVEMENT_CONTROL` falls within the scope of an interrupting event, `pickup`. When `pickup` happens, the desired behaviour would be to interrupt `L_forward`, `R_forward` and `F_forward` within their UCFs and stop the rotation of wheels immediately. As with timeout, there is yet no mechanism to do this in the framework's interface to UCFs.

Unfortunately, the two scenarios described above will not behave as expected due to the fact that UCFs are not interruptible functions at the moment. To achieve the correct behaviour of the timed operators and UCFs, additional research is needed to determine proper ways of interrupting UCFs.

5 Performance Metrics

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 release. We hypothesized that implementation of the interrupt operator in CSP++ had to add some overhead to the framework, since using exceptions forces the compiler to generate information needed to carry out stack unwinding, destructor invocation, and so on. In contrast, timeouts are simply a 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 timed operators in the new version of CSP++.

The performance comparison between untimed CSP++ (v4.2) and timed CSP++ (v5.0) was conducted in a controlled environment. All tests were performed 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 the GNU Pth version 2.0.7.

Both versions of CSP++, v4.2 and v5.0, were compiled without -DACTWATCH and -DMEMWATCH flags to avoid unnecessary output to the console, which would have increased execution time significantly.

Once compiled into C++ code, CSP specifications were run without the tracing flag “-t” or the idle check flag “-i”. The applications, however, were given the quick exit flag “-

q”, which avoids printing a dump when the system executes STOP.

Each test was run twenty-one times with the average of the last twenty being used for comparison. The first run was discarded to account for the effect of paging. Execution times were obtained using linux ‘time’ command. The sum of the user and system times was used for comparison. The largest standard deviation for any group of twenty 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 W.Gardner [Gar00], which has become a 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.

In this chapter we will examine three questions: (1) Did the addition of the timing operators in CSP++ v5.0 impact its performance when compared to CSP++ v4.2? (2) Does inclusion of an interrupt operator influence the performance of a given specification? (3) Does the use of exceptions have an impact on the memory size of the compiled program?

5.1 Untimed and 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 the comparison is intended to reveal any increased burden on the execution time as a result of exceptions support in v5.0.

1. 2 interleaved clients performing 10,000 requests

```
C(1,n) = if n>0 then ds!1.100 -> ack.1 -> C(1, n-1)
          else SKIP
```

```
C(2,n) = if n>0 then ds!2.150 -> ack.2 -> C(2, n-1)
          else SKIP
TEST(i) = ((C(1,i) ||| C(2,i));STOP
SYS = (DSS [|{|ds,ack|}|] TEST(5000)) \ {|dint, dio|}
```

2. 4 interleaved clients performing 20,000 requests

```
C(1,n) = if n> 0 then ds!1.100 -> ack.1 -> C(1,n-1)
          else SKIP
...
C(4,n) = if n> 0 then ds!4.350 -> ack.4 -> C(4,n-1)
          else SKIP
TEST(i) = ((((C(1,i) ||| C(2,i)) ||| C(3,i)) ||| C(4,i));
          STOP
SYS = (DSS [|{|ds,ack|}|] TEST(5000)) \ {|dint, dio|}
```

3. 8 interleaved clients performing 40,000 requests

```
C(1) = if n> 0 then ds!1.100 -> ack.1 -> C(1,n-1)
else SKIP
...
C(8) = if n> 0 then ds!8.950 -> ack.8 -> C(8,n-1)
else SKIP
TEST(i) = ((((C(1,i) ||| C(2,i)) ...||| C(8,i));STOP
SYS = (DSS [|{|ds,ack|}|] TEST(5000)) \ {|dint, dio|}
```

`C(i)` represents client processes that send requests to the disk server. The disk server, in return, sends an acknowledgment. There was no output to `stdout` in order to cut execution time, and concentrate solely on the system's performance.

Table 5-1. Performance of CSP++ v5.0 and v4.2

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

As we can observe from the test results, CSP++ v4.2 is slightly faster: by 0.04 seconds in the first test, 0.42 seconds in the second test, and 0.91 seconds in the third test. Our original hypothesis was that the difference in execution time was due to the use of 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 gcc **-fno-exceptions** flag produced no appreciable difference. Therefore, the difference can be attributed mainly to the interrupt operator's related actions performed in the framework. In CSP++ v5.0, when a new `Agent` is created, its constructor checks to see if it is within the scope of an interrupt operator. If it is, it copies its parent's pointer to the relevant `EnvInt` object, and if not, sets its pointer to `NULL`. Furthermore, every blocking function in the framework, upon wake up from sleep, checks to see if the wake up call was due to interrupt. This marginal additional overhead in the framework is borne by all specifications whether they use interrupts or not.

The test results show that the difference in time increases with the number of parallel processes and requests to the disk server. Each request to the disk server participates in synchronization, falling under the category of blocking functions, which perform extra checks not present in the untimed version of CSP++. Nonetheless, the net effect is very slight.

5.2 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 10000 disk requests:

1. The specification did not include the interrupt operator:

```
C(1,n) = if n>0 then ds!1.100 -> ack.1 -> C(1, n-1)
          else SKIP
C(2,n) = if n>0 then ds!2.150 -> ack.2 -> C(2, n-1)
          else SKIP
TEST(i) = ((C(1,i) ||| C(2,i));STOP
SYS = (DSS [|{|ds,ack|}|] TEST(5000)) \ {|dint, dio|}
```

2. The specification included the timed interrupt operator, but the timing was set so that the specification had sufficient time to execute and successfully terminate before the interrupt:

```
C(1,n) = if n>0 then ds!1.100 -> ack.1 -> C(1, n-1)
          else SKIP
C(2,n) = if n>0 then ds!2.150 -> ack.2 -> C(2, n-1)
          else SKIP
TEST(i) = ((C(1,i) ||| C(2,i));STOP
INTER = (DSS [|{|ds,ack|}|] TEST(5000)) \ {|dint, dio|}
SYS = INTER /20\ (interrupted -> SKIP)
```

Table 5-2. CSP++ v5.0 impact of interrupts

	No interrupt operator	**Interrupt does not happen**
v5.0	11.77 sec	12.51 sec

Table 5-2 presents results of the two tests. As can be seen the mere inclusion of the interrupt operator increases execution time by 0.74 seconds. In this case, interleaved processes C(1,i) and C(2,i) will find themselves within the scope of the interrupt operator, meaning they will have to put themselves on the `EnvInt` object's waiters list. Furthermore, the `startTI()` function will be triggered, starting process INTER as a separate thread of execution, and, upon wake up from sleep, checking whether the interrupt happened or INTER finished its execution successfully.

5.3 Memory cost of using exceptions

As shown above, exceptions did not add significant overhead to execution. Nevertheless, the gcc manual stated that the use of exceptions may “significantly” increase data size. To see exactly how much overhead the exceptions produced, we generated the DSS case study (first version mentioned in section 5.1 with 2 interleaved processes and 10000 disk requests) using three compiled versions of the CSP++ framework:

1. v4.2 compiled with gcc **-fno-exceptions** flag
2. v4.2 compiled with gcc **-fexceptions** flag
3. v5.0 compiled with gcc **-fexceptions** flag

Note that **-fexceptions** is the present default for gcc 4.

Table 5-3. DSS executable sizes with and without exceptions

	v4.2 **-fno-exceptions**	**v4.2** **-fexceptions**	**v5.0** **-fexceptions**
size (KB)	258.7	261.5	288.3

As can be seen from Table 5-3, 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 much greater and can be attributed to the addition of timed operators. This is reasonable since adding more features leads to greater framework size.

In conclusion, contrary to our expectations, the additional overhead attributable to the support for the new timed operators is minimal.

6 CSP++ and CSP libraries

Communicating Sequential Processes was never intended to be a programming language. It does, however, provide the necessary features and constructs to describe the design of concurrent systems, and, more importantly, formally verify their correctness. A number of projects have been focusing on implementing the CSP formalism in a popular programming language. Chapter 2 gave a brief survey of the existing projects. In this chapter we will compare CSP++ with projects developed at the Computing Laboratory of the University of Kent, England, namely, C++CSP2 [Bro07a] and JCSP [WBM+07]. Unlike CSP++, the two libraries do not feature automatic code generation, however, they implement a range of CSP constructs aimed at easing concurrent programming, and can be compared to CSP++'s back-end framework. The goal of this study was to see how a programmer may use CSP++, JCSP and C++CSP2. It also gave us a chance to see if CSP++ was on-par with other CSP-related programming tools, and consider possible directions for future CSP++ research and development.

C++CSP2 and JCSP share the same API and have similar structure, however, being targeted toward two different programming languages, the libraries do have distinct features. The review will focus on JCSP and provide examples in Java, however, features distinct to each project will be outlined. Appendix C includes CSP and CSP++ generated C++ code for all of the examples presented in this chapter.

6.1 JCSP and C++CSP2 Features

JCSP is a Java class library implementing a range of ideas from Communicating Sequential Processes and π-calculus [Mil99]. It was first developed by Paul Austin at the University of Kent, England, in 1997. It has since been reworked and extended, and is currently at version 1.1. JCSP was designed to provide an alternative concurrency model for Java. JCSP does not provide a completely separate concurrency model, but rather builds on top of built-in Java threads and monitors. [Wel98, Han99] examined Java's built-in concurrency model, and argued that, while the concept of monitors is easy to understand, it does not scale well and does not guarantee livelock-free and starvation-free situations for synchronizing processes (due to Java's implementation of `wait()` and `notify()` methods). JCSP was the forerunner of C++CSP library, thus they share the same API.

Working with JCSP or C++CSP2 does not require any familiarity with CSP, and knowledge of the API would be sufficient to implement a project. However, if starting with a formal CSP specification of a system, a programmer using JCSP or C++CSP2 would have to be trained in CSP in order to hand translate the CSP specification into code linked to these libraries, as there is no translator comparable to that of CSP++. JCSP and C++CSP2 are not synthesis tools like CSP++, thus, in order to use CSP alongside these libraries, hand-translation of CSP specifications into code will be necessary.

In the following subsections we will inspect the implementation of CSP constructs in the JCSP library. We will start the review by looking at CSP process composition, communication, and synchronization. We will continue with choice situations and communication via alternative channels. Finally, we will conclude with the implementation of CSP timing constructs.

6.1.1 Processes

A CSP process can be viewed as a self-contained entity that encapsulates data structures and algorithms that perform operations on these data structures. All communication between processes—except for parameters passed when a process is invoked—is done solely through channels (CSP primitives specifically designed for process communication). Following these assumptions, each CSP process in JCSP is an object of a class implementing the `CSProcess` interface. Actions performed by each object are defined in its `run()` method, which is similar to Java's approach.

Below is the general structure of a CSP process implemented in JCSP:

```
import org.jcsp.lang.*;
...  other imports

 class ProcessExample implements CSProcess {

   ...  private/protected shared synchronisation objects (channels, etc.)
   ...  private/protected state information

   ...  public constructors
   ...  public configuration/inspection methods (when not running)

   ...  private/protected support methods (part of a run)
   ...  public run method (the process starts here)
}
```

JCSP implements process-oriented design, which consists of concurrently executing processes communicating with each other through a set of synchronization objects. `CSProcess` public constructors handle installation of the synchronization objects into the process's private/protected fields. `set()` and `get()` methods may be invoked to change the configuration of any process, however, this can only be done in between `run()`s and only by a single process, usually the parent process, otherwise, unpredictable behaviour may arise when a process's state is changed by anything other than channel communication or synchronization.

Consider a very simple countdown process written in CSP. Process COUNTDOWN performs event a.*i* (*i*=10..1) 10 times, then event blast_off, and successfully terminates:

```
COUNTDOWN(n) = if n > 0
                    then a.n → COUNTDOWN(n-1)
                    else blast_off → SKIP
COUNTDOWN(10)
```

This example incorporates the CSP concept of "looping" via recursion, and produces the trace <a.10, a.9, ..., a.1, blast_off>. When written in JCSP, the above example will look like this:

```
public class Countdown implements CSProcess {

      final private Integer N;

      public Countdown (final int n) {
            this.N = new Integer (n);
      }

      public void run () {
            while (N > 0) {
                  System.out.println(N);
                  N--;
            }
            System.out.println("blast_off");
      }
}
```

And the program driver, which starts the execution:

```
class CountdownTest {

      public static void main (String[] args) {
            Countdown test = new Countdown (10);
            test.run();
      }
}
```

It prints 10, 9, ..., 1 on successive lines and then blast_off, equivalent to the trace.

C++CSP2 follows a similar library structure. Every executing process is typically subclassed from `CSProcess`. The actual work of every process is included in the `protected void run()` function. Each `CSProcess` is allocated on the heap by the programmer. C++CSP2 manages deletion of processes once they finish running. C++CSP2 warns against manual deletion of `CSProcess`es or their allocation on the stack, which may cause memory leaks.

6.1.2 Process Composition

Concurrent execution of processes is achieved through the use of the `Parallel` class. An array of `CSProcess`es can be passed to the `Parallel` constructor. The constructor, in turn, returns an instance of `CSProcess` that represents the parallel composition of its process arguments. Execution of `Parallel` `CSProcess` terminates when all of its component processes finish their runs. `CSProcess`es can be added to parallel execution either by supplying instances of `CSProcess`es as arguments to the `Parallel` constructor or by use of the `addProcess()` method. If an attempt is made to add a process while `Parallel` is executing the `run()` method, the additional processes will be added after the completion of the current `run()` but before its next invocation, if the parallel composition is to be run again.

Let us expand the COUNTDOWN example presented in the previous section. This time we create two processes, each of which performs a countdown completely separately from the other process. Such composition would correspond to the CSP concept of interleaving, where process run in parallel but do not communicate or synchronize on any events:

ROCKET1 = COUNTDOWN(10)

ROCKET2 = COUNTDOWN(10)

ROCKET1 ||| ROCKET2

The JCSP implementation of processes COUNTDOWN will be as before, however, the driver will undergo some changes:

```
class CountdownTest {

      public static void main (String[] args) {
            Countdown rocket1 = new Countdown (10);
            Countdown rocket2 = new Countdown (10);
            new Parallel (rocket1, rocket2).run();
      }
}
```

Alternatively, to produce sequential composition of the above processes—ROCKET1; ROCKET2—where `rocket1` is launched first and `rocket2` second, the `Parallel` instance would simply be changed to `Sequence`:

```
class CountdownTest {

      public static void main (String[] args) {
            Countdown rocket1 = new Countdown (10);
            Countdown rocket2 = new Countdown (10);
            new Sequence (rocket1, rocket2).run();
      }
}
```

Process composition follows the same rules and similar syntax in C++CSP2. Its major difference is the ability of a programmer to decide whether to run `Parallel` processes in one user-level thread or different user-level threads to take advantage of multicore and multiprocessor systems. This is discussed in more detail later in the chapter.

6.1.3 Channel communication

As mentioned earlier, a `CSProcess` can communicate with other `CSPprocess`es solely through the use of synchronization objects, and not by calling each other's public methods. This follows the idea of CSP channels. Recall that a CSP channel is a unidirectional non-buffered fully-synchronized means of communication between processes.

In JCSP, the `Channel` class implements CSP channel semantics. It is a factory with static methods for different kinds of channels capable of passing different types of objects. JCSP provides the traditional CSP non-buffered fully synchronizing channels as well as a number of buffered channels [WBM+07]. All channel types can be summarized into four categories: `One2OneChannel` connecting one writing process with one reading process; `Any2OneChannel` can have multiple writers, but only one reader; `One2AnyChannel` supports one writer and a one reader, but the reader can be any process from a list; and `Any2AnyChannel`, where multiple writers share the channel with multiple readers, however, at any one time, the communication still occurs between one writer and one reader. A full list of the types of channels available in JCSP can be found at [AW].

After a new instance of a channel is created, the appropriate channel ends are passed to corresponding `CSProcess`es. Having references to specific channel ends prevents a reader process from writing to a channel, and a writer from reading. Channel ends mirror similar concepts in the occam-pi language, and the authors of JCSP argue that they introduce increased safety into process communication [WBM+07].

Consider a simple process communication example. Process `WRITER` outputs 2 to channel `a`, while process `READER` reads the contents of the channel into variable `x`.

$$\text{WRITER} = a!2 \rightarrow \text{SKIP}$$
$$\text{READER} = a?x \rightarrow \text{SKIP}$$
$$\text{READER} \;||_a\; \text{WRITER}$$

JCSP implementation of the above scenario will look like this:

```
public class Writer implements CSProcess {

      final private ChannelOutput out;

      public Writer (final ChannelOutput out) {
            this.out = out;
      }

      public void run () {
            out.write(2);
      }
}

public class Reader implements CSProcess {
      final private Integer N;
      final private ChannelInput in;

      public Writer (final ChannelInput in) {
            this.in = in;
      }

      public void run () {
            N = in.read();
      }
}

class Driver {

      public static void main (String[] args) {

            final One2OneChannel a = Channel.one2One ();

            new Parallel (    new Writer (a.out()),
                              new Reader (a.in()) ). run();
      }
}
```

The driver first creates a simple `One2OneChannel`. Then it creates two processes and passes each one the appropriate channel end. Finally, it creates a `Parallel` instance to run the processes.

More complex channel synchronization can include multiple writers and a single reader, single writer and multiple readers, or even multiple writers and multiple readers. In JCSP this is accomplished through the use of shared channels. Synchronization still occurs only between pairs of processes, a single reader and a single writer, though multiple readers and/or writers may share the channel by taking turns. Shared channel users take turns on a first-come-first-served basis, which should not be confused with broadcasting, where data from a single writer may be read by numerous readers.

JCSP includes two major types of channels, all of which support `one2one`, `any2one`, `any2any`, and other interfaces:

- Object Channels capable of passing any Java `Object`s. `ChannelInt` class has become deprecated and Object Channel should be used, however, its remains are still seen when constructing choice situations with the `Alternative` class (described further in the chapter).
- CALL Channels provide mechanisms for client-server communication, where the client "calls" the server and, upon accept, invokes one of the methods provided by the server's interface.

C++CSP2 follows the same design of channel communication with minor differences in syntax.

6.1.4 Process Synchronization

When several processes need to synchronize on a particular event, JCSP employs the concept of barriers, which is implicit in CSP synchronization. A process participating in event synchronization cannot cross the barrier until all processes participating in the synchronization arrive at the barrier and cross it together. A process arriving at the synchronization point before other processes will simply block, awaiting wake up when all synchronizing processes arrive. Once a process reaches the synchronization point, or the barrier, it cannot back off and is committed to synchronizing. Consider a CSP example of four abstract processes synchronizing on event a:

PROC(i) = at.i → a → over.i → SKIP

$((\text{PROC}(1)\ ||_a\ \text{PROC}(2))\ ||_a\ \text{PROC}(3))\ ||_a\ \text{PROC}(4)$

A valid trace would be <at.1, at.2, at.3, at.4, a, over.1, over.2, over.3, over.4>, or with any permutation among the at's and over's.

JCSP implementation of the above specification will be the following:

```
public class Process implements CSProcess {

      private final Barrier a;

      public Process (int id, Barrier a) {
            this.id = id;
            this.a = a;
      }

      public void run () {
            System.out.println ("Proc " + id + " at the barrier");
            a.sync ();
            System.out.println ("Proc " + id + " over the barrier");
      }
}

public class Driver {

      public static void main (String[] args) {

            final int nProcs = 4;
            final Barrier a = new Barrier (nProcs);
            final Proc[] procs = new Proc[nProcs];

            for (int i = 0; i < procs.length; i++) {
                  procs[i] = new Process (i, a);
            }

            new Parallel (procs).run ();
      }
}
```

C++CSP2 implements similar design of barriers to handle process synchronization.

6.1.5 Choice

In CSP, there are two types of choice: deterministic (external) and non-deterministic (internal). We will focus on deterministic choice since it is more useful in software design. Consider a simple CSP process below:

$P = a \rightarrow SKIP \; \Box \; b \rightarrow SKIP$

$Q = a \rightarrow SKIP$

$P \; ||_{a,b} \; Q$

Process P is offering to engage in either event a or b, however, process Q can only perform event a. Since P and Q are synchronized on both events, Q's performance of a determines P's execution as well.

JCSP's `Alternative` class implements CSP semantics of choice. In essence it offers several events for synchronization with other `CSProcess`es, and waits until one of the offers is taken by the synchronizing `CSProcess`. Events that are offered by a `CSProcess` in a choice situation are known as Guards in JCSP, as they guard the process's execution beyond the choice. If the choice is not made by a synchronizing party, Guards will not let the `CSProcess` proceed on its own.

The `Alternative` constructor takes an array of guards, events offered for synchronization, and returns as object. When the running process desires to make the choice, it calls `select()` on the `Alternative` object, which returns the index of the chosen guard. There are six types of guards that `Alternative` can handle, or six types of events that a `CSProcess` can offer in a choice situation:

- `AltingChannelInput`: object channel input. Ready if unread data is pending in the channel.
- `AltingChannelInputInt`: integer channel input. Ready if unread data is pending in the channel.
- `AltingChannelAccept`: CALL channel accept. Ready if an unaccepted call is pending on the channel.

- `AltingBarrier`: barrier synchronization. Ready if all enrolled processes are offering to synchronize.
- `CSTimer`: timeout. Ready if the timeout has expired (timeout values are absolute time values, not delays).
- `Skip`: skip. Always ready.

Each of the `Alting*` types of guard have to be created in the driver class, because they are essentially “global” to the program. The `Alting*` objects are passed as parameters to the processes that need to use them in constructing an `Alternative`.

JCSP implements different ways of resolving the choice situation in cases where more than one guard is ready:

- `select`, already mentioned, makes an arbitrary choice between guards, if more than one is ready.
- `priSelect`: if more than one of the guards is ready, it chooses the first one in the list.
- `fairSelect` If more than one of the guards is ready, the chosen guard is assigned a priority, so that on the next iteration of `fairSelect` with the `Alternative`, it has the lowest priority. This ensures that no guard is serviced more than once before all guards are serviced.

Note that each select method will block, suspending the calling thread, until one of the guards becomes ready and the choice situation can be resolved, whereupon it returns the index of the chosen guard.

After reviewing `Alternative` features, let us consider the JCSP implementation of the CSP scenario presented at the beginning of this section. This simple choice is an application for the `AltingBarrier` type of guard:

```
public class P implements CSProcess {
      private final AltingBarrier a;
      private final AltingBarrier b;

      public Process (AltingBarrier a, AltingBarrier b) {
            this.a = a;
            this.b = b;
      }

      public void run () {
            final Alternative choice =
                  new Alternative (new Guard [] {a, b});

            final int index = choice.select();

            if(!index) {
                  a.sync();
                  System.out.println(“Synced on event a“);
            }
            else {
                  b.sync();
                  System.out.println(“Synced on event b”);
            }
      }
}

public class Q implements CSProcess {

      private final AltingBarrier a;

      public Process (AltingBarrier a) {
            this.a = a;
      }

      public void run () {
            a.sync();
            System.out.println(“Synced on event a“);
      }
}

public class Driver {

      public static void main (String[] args) {

            final AltingBarrier a = new AltingBarrier (2);
            final AltingBarrier b = new AltingBarrier (2);

            P p = new P(a, b);
            Q q = new q(a);

            new Parallel (p, q).run ();
      }
}
```

Note that two AltingBarrier guards—one for event a and one for event b—are created in Driver for use in P and Q, each barrier accommodating two parties (P and Q) as given by the constructor parameter (2). The guard objects are passed to P and Q when they are created. P, who does the choice, uses them to create an Alternative and then calls select() on it. Q, who does not do a choice, simply calls sync() on its barrier to signal its arrival. After P decides which event is ready, it still must call sync() to complete the synchronization. After that, both P and Q may proceed independently (here, they simply terminate).

The program given above illustrates an interesting point. Implementing a trivial CSP specification involves a high ratio of complex JCSP code. Synthesized CSP++ code will have a comparable number of lines of C++, however, all that code is automatically generated, thus eliminating possibilities for human programming errors. If the CSP specification is correct, then the synthesized code is "correct by construction."

Alternative allows implementation of a polling action similar to the one of the untimed timeout in CSP++. We want to see if processes are ready to synchronize on an event or channel right away. If not, an alternative action can be taken. In JCSP this can be accomplished by using priSelect between an event and a skip guard (placed in the guard array after the event to signify lower priority). Below is the implementation of such a scenario, where we poll against three channels to see if there is any data pending in them. If not, the program "times out" in favour of something else:

```
public class Polling implements CSProcess {

      private final AltingChannelInput in0;
      private final AltingChannelInput in1;
      private final AltingChannelInput in2;
      private final ChannelOutput out;
      public Polling (final AltingChannelInput in0,
                      final AltingChannelInput in1,
                      final AltingChannelInput in2,
                      final ChannelOutput out) {
            this.in0 = in0;
            this.in1 = in1;
            this.in2 = in2;
            this.out = out;
      }

      public void run() {

            final Skip skip = new Skip ();
            final Guard[] guards = {in0, in1, in2, skip};
            final Alternative alt = new Alternative (guards);

            switch (alt.priSelect ()) {
                  case 0:
                  ...  process data pending on channel in0 ...
                  break;
                  case 1:
                  ...  process data pending on channel in1 ...
                  break;
                  case 2:
                  ...  process data pending on channel in2 ...
                  break;
                  case 3:
                  ...  nothing available for the above ...
                  ...  so get on with something else for a while ...
                  break;
            }
      }
}
```

C++CSP2 does not provide any additional features or differ significantly in its implementation of choice constructs.

6.1.6 Timing Constructs

Some of the CSP timing constructs seamlessly integrate into JCSP library. For example, the timed prefix operator in Timed CSP specifies the amount of time that has to pass between the finish of one event and attempt of the next. This operator was not mentioned explicitly in the API documentation, however, was included in several examples. Consider JCSP code below, that corresponds to

$$\text{DelayExecution} = a \xrightarrow{5} b \rightarrow \text{SKIP}$$

in CSP (considering one-second granularity for time units):

```
public class DelayExecution implements CSProcess {

      private final Barrier a;

      public Process () {
      }

      public void run () {
            final CSTimer tim = new CSTimer();
            System.out.println ("executing abstract event a");
            tim.sleep(5000);        //sleeping for 5 seconds
            System.out.println ("executing abstract event b");
      }
}
```

The `CSTimer` class in JCSP provides the necessary timing constructs. In the example above `sleep()` method simply puts the current thread of execution to sleep for a specified amount of time, and has the same semantics as `java.lang.Thread.sleep`.

More complex cases including `CSTimer` may be constructed. Consider the following Timed CSP specification involving a timed timeout:

$$\text{TIMEOUT} = a \rightarrow \text{SKIP} \overset{30}{\triangleright} \text{tout} \rightarrow \text{SKIP}$$

We can implement the specification in JCSP using the `Alternative` class and the `priSelect()` method.

```
public class TIMEOUT implements CSProcess {

      private final AltingBarrier a;

      public Process (AltingBarrier a) {
            this.a = a;
      }

      public void run () {
            final CSTimer tim = new CSTimer();
            tim.setAlarm(tim.read() + 30000);
            final Guard[] guards = {tim, a};
            final Alternative alt = new Alternative (guards);
            final int index = alt.priSelect();
            if(index)
```

```
                System.out.println(“Handling event a”);
            else
                System.out.println(“Timeout: 30 secs elapsed”);
        }
}
```

In the example above `Alternative` `priSelect()` method has to be used while the `CSTimer` instance can be placed in the guard array before or after the other event. `CSTimer` placement in the guard array will determine which event is favoured upon thread wake up if both were ready. In the example above the timeout will be favoured.

Unfortunately, JCSP does not provide interrupt facilities which map directly to CSP, where, for example, the initial event of process A interrupts process B and no further events from B or its subprocesses show up in the execution trace. However, in JCSP it is possible to use `Alternative` and `CSTimer` to come up with interesting examples that do resemble interrupting behaviour, but are far from CSP semantics of the interrupt.

The example below reads in data from the input channels for a user-defined amount of time. Once the time elapses the process abandons reading from the input channels and goes to do something else. This resembles Timed CSP’s timed interrupt to some extent, when a process is given so much time to perform its duties, and if it does not complete in time its execution is interrupted in favour of another process. In the example below, however, the process does complete each channel input on every iteration; the interrupt is only serviced before the next channel input iteration.

```
public class FairPlexTime implements CSProcess {

    private final AltingChannelInput[] in;
    private final ChannelOutput out;
    private final long timeout;

    public FairPlexTime (final AltingChannelInput[] in,
                  final ChannelOutput out,
                  final long timeout) {
        this.in = in;
```

```
            this.out = out;
            this.timeout = timeout;
      }

      public void run () {
            final Guard[] guards = new Guard[in.length + 1];
            System.arraycopy (in, 0, guards, 0, in.length);

            final CSTimer tim = new CSTimer ();
            final int timerIndex = in.length;
            guards[timerIndex] = tim;

            final Alternative alt = new Alternative (guards);

            boolean running = true;
            tim.setAlarm (tim.read () + timeout);
            while (running) {
                  final int index = alt.fairSelect ();
                  if (index == timerIndex) {
                        running = false;
                  } else {
                        out.write (in[index].read ());
                  }
            }
      }
}
```

Even though the example above is not a true interrupt, it does provide the programmer with a useful timing tool.

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

6.2 CSP++ vs. JCSP and C++CSP: Pros And Cons

Now that the reader has some familiarity with CSP++ as well as JCSP and C++CSP2, we can analyze some of the advantages and disadvantages of these different approaches to implementing CSP constructs alongside a popular programming language. Moreover, we can see if CSP++ can benefit from some of the design decisions present in JCSP and C++CSP2. The areas discussed below are the underlying threading mechanism, possibilities for formal verification, and support for networking.

6.2.1 Threading Libraries

Communicating Sequential Processes was specifically designed to describe concurrent systems. A threading mechanism must be used to simulate concurrency. CSP++, JCSP, and C++CSP2 are all based on different threading models, and each offers its own advantages.

The JCSP project runs on top of Java Virtual Machine (Sun's distribution of JVM on most operating systems) and uses kernel-space threads. Kernel-space threads usually rely on thread preemption for context switches that are performed by the operating system kernel without the application's knowledge. Kernel-space threads perform context switches slower than user-space threads, however, they have a big advantage of benefitting from multi-core and multiprocessor systems by potentially scheduling each executing thread on a separate processor (provided that the number of cores or processors outnumbers threads in the ready queue).

CSP++ uses the non-preemptive user-space threading library, GNU Pth. Being implemented completely in user space, Pth performs context switches much faster than kernel-space threading mechanisms. Another advantage of Pth in particular, is the fact that it does not require any platform-dependant assembly manipulation, which leads to high portability. The biggest drawback of user-space threading libraries, however, is their inability to benefit from multi-core and multiprocessor systems.

C++CSP was originally based on user-space threading libraries, but Version 2 moved to a home-brewed hybrid model, a mixture of user-space and kernel-space threading mechanisms, to benefit from both worlds. Multiple kernel-space threads may be running in parallel, while each kernel-space thread may contain multiple user-space threads. C++CSP2 implements a C++CSP-kernel in each kernel-space thread. C++CSP-kernel maintains a run queue and a timeout queue. Each `CSProcess` uses one user-space thread, and each user-space thread always resides in the same kernel-space thread.

In a multi-core or multiprocessor system, C++CSP2 gives a programmer the ability to choose how to run `CSProcess`es: in the same kernel-space thread or in different kernel-space threads. By default all `CSProcess`es are run in a new kernel-space thread, however, process composition structures like `RunInThisThread`, `InParallelOneThread`, and `InSequenceOneThread` allow the programmer to force a `CSProcess` to be run in the same kernel-space thread. Of course, the ability to benefit from multi-core and multiprocessor systems came with a cost — assembly hacking in Windows and Linux environments to tailor C++CSP2 versions specifically to each operating system.

6.2.2 Formal Verification

Besides providing useful constructs and assumptions when reasoning about concurrent systems, CSP has a built-in formal verification model that allows a system designer to formally verify the correctness of his or her design. Automated tools such as FDR2 and ProBE help the system designer explore the state space of their designs as well as verify them against deadlocks, livelocks, and other race hazards. CSP++, JCSP and C++CSP2 differ significantly in the way formal design verification is done.

What sets CSP++ apart from JCSP and C++CSP2 is the fact that the system design can be formally verified before the CSP++ tool is utilized and the corresponding C++ code is generated, and afterwards, to confirm that the implementation refines the design. What makes this possible is the fact that CSP++ is a software synthesis tool. A system architect starts by making a system design in CSP. The design is run through ProBE and FDR2 tools until the designer is confident that the system is free of deadlocks, livelocks, etc. The design is then synthesized using CSP++, and C++ code is generated with synchronizations and channel communications already in place. user-coded functions may be linked to abstract CSP events. If user-coded functions do not interfere with process synchronization and communication produced by CSP++, the generated code will correspond to the original CSP design in that it will be free of deadlocks, livelocks, and other race conditions. This can be checked by running the execution trace produced by the generated C++ code through FDR2 to see if it refines the original CSP specification.

JCSP's implementation of channels and `Alternative` has been formally verified and the results described in detail in [WM00]. C++CSP2 also refers to this verification study to prove the correctness of its implementation, as the libraries share the same API and design. Unfortunately, the fact that the library implemented these constructs correctly does not prove that a programmer writing a system with JCSP and C++CSP2 will not make mistakes setting up channel communication or event synchronization. Consider this example in C++CSP2 [Bro07b]:

```
class NumberSwapper : public CSProcess
    {
    private:
        csp::Chanin<int> in;
        csp::Chanout<int> out;
    protected:
        void run()
        {
            int n = 0;
            while (true)
            {
                out << n;
                in >> n;
            }
        }
    public:
        NumberSwapper(const Chanin<int>& _in,const Chanout<int>& _out)
            :   in(_in),out(_out)
        {
        }
    };
```

And a driver program that composes two `NumberSwapper`s in parallel:

```
One2OneChannel<int> c,d;
    Run(InParallel(
        ( new NumberSwapper( c.writer(), d.reader() ) )
        ( new NumberSwapper( d.writer(), c.reader() ) )
    );
```

The outcome of this program is a deadlock. Both `NumberSwapper`s are trying to write to a channel at the same time. Even though the channels they are using are different, reading from those channels never occurs, as each instance is waiting for the data to be read from each channel, so channel synchronization never occurs. C++CSP2 provides some error detection mechanism in the form of `DeadlockError` exception thrown in the driver program. The exception is thrown when all `CSProcess`es are blocked within the C++CSP2 system, i.e., unresolved synchronization or channel communication. According to C++CSP2 documentation [Bro07b] the `DeadlockError` is fatal and unrecoverable. However, having at least some exception handling does ease the debugging process. It should be noted that only C++CSP2 provides this type of exception handling; JCSP does not.

In CSP++ such deadlock situation can only occur if the system designer did not sufficiently check the CSP specification with FDR2 (or if UCFs erroneously interfered with synchronization already set up by CSP++). Nevertheless, to battle such situations, CSP++ provides a way to detect deadlock situations. When executing a synthesized C++ program, an optional CSP++ command line flag ("`-i`", which stands for "idle") starts an idler task, which periodically checks the status of all non-terminated CSP++ processes. If all non-terminated tasks remain idle not making any progress, the program is terminated and a task status dump is performed [Gar00].

6.2.3 Networking

JCSP .net [AW] and C++CSP v1 [BW03, Bro07a] have built-in networking solutions in the sense that channel communication can be set up through sockets across networked computers. A driver program, which starts the execution, keeps track of IP addresses and ports of communicating processes. As long as correct channel ends are handled correctly, network channels do not change overall system behaviour.

CSP++ does not provide any explicit networking support at the moment, but UCFs are free to utilize socket communication, as demonstrated in the ATM case study [DG05].

6.2.4 General Thoughts

CSP++, JCSP and C++CSP2 are all based on CSP and provide useful concurrency tools and constructs in popular programming languages, but, in the end, are different projects targeted toward different end users.

CSP++ can be regarded as a higher-level tool when compared to JCSP and C++CSP2, as it encompasses overall system design. It does require knowledge of CSP, and thus is more suitable for use by system architects, who will develop an overall system design and formally verify its correctness. Once the CSP specification is sufficiently verified and the system “skeleton” is synthesized with CSP++, user-coded functions can be added to form the body of the developed system. UCFs do not require any knowledge of CSP and may be coded by ordinary programmers who simply implement algorithms in C++ and link them to the skeleton.

JCSP and C++CSP2 provide a rich arsenal of programming tools, although, not all of them adhere to CSP ideas, such as buffered channels, for instance. This is not a bad thing, but a CSP designer should be aware of this. These two projects have powerful features, such as networking support, Windows and Linux versions, as well as support for multi-core and multiprocessor architectures. JCSP and C++CSP2 do not require any CSP knowledge (although it was extremely useful when trying to understand their API) and may be targeted at casual programmers who are looking for alternative concurrency models. Unfortunately, the final design of programs written in JCSP or C++CSP2 cannot be automatically verified, so a programmer should be extremely careful when setting up channel communication or synchronization to avoid deadlock, or worse yet, livelock and other race conditions (as there is no exception detection mechanism for these cases).

Finally, when working with JCSP or C++CSP2 libraries, the programmer is responsible for creating and connecting static objects that mirror concurrent execution before the modelled system is run, i.e., set up parallel or sequential execution, work out synchronization and communication mechanisms, etc. In contrast, objects involving choice are constructed and evaluated during `run()`s, not beforehand. Thus, different JCSP objects require separate treatment by the programmer, depending on which category they belong to. This may quickly lead to messy and error-prone programming.

7 Conclusions and Future Work

In this thesis, we presented an updated CSP++ that was enhanced with the Timed CSP operators and their untimed counterparts. The cspt translator and the object-oriented application framework of CSP++ were extended to handle the newly-added operators. We demonstrated the new features of CSP++ by walking through the implementation details and presenting a new VAC case study. Although VAC is not directly verifiable at the moment, we showed the possibilities of future integration with HORAE. We compared CSP++ with JCSP and C++CSP2 to confirm that our tool is on-par with other CSP-inspired libraries. In the following sections we will summarize our results and present possible research directions for future work.

7.1 Conclusions

Extending CSP++ with Timed CSP operators enabled us to create a more powerful software synthesis tool suitable for soft real-time applications. Newly-added operators allowed us to synthesize a larger subset of the CSP language. More complex formal specifications can now be constructed and synthesized. We showed that often timing information can be an integral part of a software system. Now programmers can benefit from the use of timeouts, interrupts, and delays when designing a system with CSP++. We presented the VAC case study which combines all the newly added operators within one specification and shows the expressive power of our tool.

The addition of timing did not change CSP++'s integration with the commercial formal verification tools such as FDR2 and ProBE. On the contrary, in addition to all the previously available CSPm operators, we extended CSP++ with two more, the untimed timeout and the untimed interrupt. CSP programmers can remain confident in their formal specifications, while having more tools to operate with.

Unfortunately, FDR2 and ProBE cannot verify all the operators added to CSP++. Formal specifications that include timed interrupt, timed timeout, and timed prefix cannot be verified with these tools. Fortunately, we have found an additional tool capable of reasoning about timed specifications. Though HORAE is not complete, we were able to show that our tools are compatible. Despite minor changes in syntax, CSP++ was able to synthesize the correct code for three small case studies provided by the HORAE team: Dining Philosophers, Timed Vending Machine, and Timed Railroad Crossing.

We conducted a review of competing CSP-inspired libraries, JCSP and C++CSP2. The review enabled us to gain more confidence in our tool, as JCSP and C++CSP2 are two of the most developed CSP tools. CSP++, using UCFs, is capable of describing a similar range of problems as the two libraries. We also argued that using our tool allows for fewer possibilities for errors in an implementation, given a correct specification, as CSP++ takes advantage of direct formal verification of CSP specifications and automatic code generation. Nevertheless, the comparison gave us a chance to see what areas of CSP++ need more work.

7.2 Future Work

A number of interesting research opportunities still remain in CSP++:

Support for more data types. To make CSP++ a more useful programming tool it is necessary to add support of additional data types. At present, CSP++ is able to handle integers in channel communication and parameterized processes. However, integer support is not sufficient to model complex problems. In light of object-oriented design of CSP++, it would be of great benefit to support strings and abstract objects. This limitation became more evident when comparing CSP++ with JCSP and C++CSP2, which are able to take advantage of object communication.

Addition of more data types would require significant changes to the cspt translator and the object-oriented framework. Additionally, FDR2 does not support strings or objects. Therefore, implementation of additional data types would require much research to keep specifications used for synthesis with CSP++ directly verifiable in FDR2. The HORAE team did not report on any support beyond integers, making the problem spill over to the timing world.

New threading library. CSP++ is currently built on top of non-preemptive GNU Pth threading libraries. There are many advantages to using Pth, however, its slow speed and unsuitability for embedded systems have raised concerns. These issues were already addressed in [Dox05]. Also, recent explosion in the number of cores fitted onto one CPU and their relatively inexpensive availability raises the question of multi-core support in CSP++. Concerns raised by Pth alongside the inability to take advantage of numerous cores press the issue of changing CSP++'s reliance on Pth. The big question is whether to attempt to find another third-party threading library suitable for our needs and try to port CSP++ to it, or attempt to develop an in-house threading library.

Better UCF integration. Currently, UCFs cannot be used for interprocess synchronization (this is by design), cannot be timed out or interrupted, and can only participate in deterministic choice in a limited fashion (due to no callback mechanism if the choice does not immediately succeed or fail). Further research is needed in order to relax or remove these restrictions while ensuring that UCFs do not break the formally verified control backbone synthesized by CSP++. If Pth continues as the thread library of choice, these problems may be solved by its ability to issue system calls that only block the current thread, while allowing early return due to timeout or signals. These features of its API have not been used by CSP++ to date, but may prove helpful in improving the capabilities of UCFs.

Bibliography

[Ale05] Michael Alexander. Research plan: Financial process algebra: Formal models, model checking and code generation, 2005.

[AW] P.D. Austin and P.H. Welch. SP for Java (JCSP) 1.1-rc2 API Specification [online, cited February 2008]. Available from: http://www.cs.kent.ac.uk/projects/ofa/jcsp/jcsp-1.1-rc2-doc/.

[Axo07] Axon7 Multicore Solutions. Jibu documentation [online]. December 2007 [cited December 20, 2007]. Available from: http://www.axon7.com/flx/documentation/.

[Bro07a] Neil Brown. C++CSP2: A many-to-many threading model for multicore architectures. *Communicating Process Architectures 2007*, pages 183–205, 2007.

[Bro07b] Neil Brown. C++CSP2 guide [online]. August 2007 [cited February 2008]. Available from: http://www.cs.kent.ac.uk/projects/ofa/c++csp/doc/guide1.html.

[BW03] Neil Brown and Peter Welch. An introduction to the Kent C++CSP library. *Communicating Process Architectures - 2003*, pages 139–156, 2003.

[DG05] Stephen Doxsee and W.B. Gardner. Synthesis of C++ software from verifiable CSPm specifications. In *12th Annual IEEE International Conference and Workshop on the Engineering of Computer Based Systems (ECBS 2005)*, pages 193–201, Greenbelt, MD, April 2005.

[DHSZ06] 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.

[Dox05] Stephen Doxsee. Reengineering CSP++ to conform with CSPm verification tools. Master's thesis, University of Guelph, 2005.

[DS88] Jim Davies and Steve Schneider. An introduction to Timed CSP. Technical report, Oxford University Computer Laboratory, 1988.

[DS95] 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.

[DZSH06] Jin Song Dong, Xian Zhang, Jun Sun, and Ping Hao. Reasoning about Timed CSP models. In Jayadev Misra, Tobias Nipkow, and Emil Sekerinski, editors, *FM 2006: Formal Methods*, volume 4085 of *Lecture Notes in Computer Science*. Springer, 2006.

[FGM+92] J. Fernandez, H. Garavel, L. Mounier, A. Rasse, C. Rodriguez, and J. Sifakis. A toolbox for the verification of LOTOS programs. In *Proc. of the 14th International Conference on Software Engineering (ICSE'14,* pages 246–259, May 1992.

[For] Formal Systems (Europe) Ltd. Website [online, cited 11/28/07]. Available from: http://www.fsel.com/.

[Gar00] 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.

[Gar03] W.B. Gardner. Bridging CSP and C++ with selective formalism and executable specifications. In *First ACM and IEEE International Conference on Formal Methods and Models for Co-Design, 2003*, pages 237–245. MEMOCODE'03, 2003.

[Gar05] William B. Gardner. Converging CSP specifications and C++ programming via selective formalism. *ACM Trans. on Embedded Computing Sys.*, 4(2):302–330, 2005.

[Han99] Per Brinch Hansen. Java's insecure parallelism. *SIGPLAN Not.*, 34(4):38–45, 1999.

[HJ95] M.G. Hinchey and S.A. Jarvis. *Concurrent Systems: Formal Development in CSP.* McGraw-Hill Book Company, 1995.

[Hoa78] C. A. R. Hoare. Communicating sequential processes. *Commun. ACM*, 21(8):666–677, 1978.

[LO06] Alex A. Lehmberg and Martin N. Olsen. An introduction to CSP.NET. In Peter Welch, John Kerridge, and Fred Barnes, editors, *Communicating Sequential Architectures 2006*. IOS Press, 2006. 13-30.

[LT93] Nancy Leveson and Clark S. Turner. An investigation of the Therac-25 accidents. *IEEE Computer*, 26(7):18–41, July 1993. Available from: http://courses.cs.vt.edu/ cs3604/lib/Therac_25/Therac_1.html.

[Mil99] R. Milner. *Communicating and Mobile Systems: the Pi-Calculus.* Cambridge University Press, 1999.

[Ols06] Martin Nebelong Olsen. Multithreaded programming using CSP.NET [online]. September 2006 [cited December 12, 2007]. Available from: http://www.codeguru.com/csharp/.net/net_general/toolsand3rdparty/article.ph%p/c12533/.

[OS05] Joël Ouaknine and Steve Schneider. Timed CSP: A retrospective. *This paper is electronically published in Electronic Notes in Theoretical Computer Science,* 2005. Available from: www.elsevier.nl/locate/entcs.

[Pou04] Kevin Poulsen. Tracking the blackout bug [online]. April 2004 [cited 11/28/07]. Available from: http://www.securityfocus.com/news/8412.

[RR88] G. M. Reed and A. W. Roscoe. A timed model for Communicating Sequential Processes. *Theor. Comput. Sci.*, 58(1-3):249–261, 1988.

[RR99] G. M. Reed and A. W. Roscoe. The timed failures-stability model for CSP. *Theor. Comput. Sci.*, 211(1-2):85–127, 1999.

[RRS03] V. Raju, L. Rong, and G. S. Stiles. Automatic conversion of CSP to CTJ, JCSP, and CCSP. In Jan F. Broenink, editor, *Communicating Process Architectures 2003*. IOS Press, 2003.

[Sch00] Steve Schneider. *Concurrent and Real-time Systems: The CSP Approach*. John Wiley & Sons, Ltd., 2000.

[WBM+07] 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.

[Wel98] P. H. Welch. Java Threads in the Light of occam/CSP. In P.H.Welch and A.W.P.Bakkers, editors, *Architectures, Languages and Patterns for Parallel and Distributed Applications*, volume 52 of *Concurrent Systems Engineering Series*, pages 259–284, Amsterdam, April 1998. WoTUG, IOS Press. Available from: http://www.cs.kent.ac.uk/pubs/1998/702.

[WM00] Peter H. Welch and Jeremy M. R. Martin. Formal analysis of concurrent Java systems. In *Communicating Process Architectures 2000*. IOS Press, 2000.

Appendix A: Timed CSP Case Studies

Dining Philosophers

HORAE syntax:

```
PHIL1 =  enter1 -> ((pick11 -> pick12 -> eat1 -> put11 -> put12 -> leave1 -> PHIL1)
                    []
                 (pick12 -> pick11 -> eat1 -> put12 -> put11 -> leave1 -> PHIL1))

PHIL2 =  enter2 -> ((pick22 -> pick23 -> eat2 -> put22 -> put23 -> leave2 -> PHIL2)
                    []
                 (pick23 -> pick22 -> eat2 -> put23 -> put22 -> leave2 -> PHIL2))

PHIL3 =  enter3 -> ((pick33 -> pick31 -> eat3 -> put33 -> put31 -> leave3 -> PHIL3)
                    []
                 (pick31 -> pick33 -> eat3 -> put31 -> put33 -> leave3 -> PHIL3))

CHOP1 = (pick11 -> put11 -> CHOP1) [] (pick31 -> put31 -> CHOP1)

CHOP2 = (pick22 -> put22 -> CHOP1) [] (pick12 -> put12 -> CHOP2)

CHOP3 = (pick33 -> put33 -> CHOP1) [] (pick23 -> put23 -> CHOP3)

PHILS = (PHIL1 |||  PHIL2) ||| PHIL3

CHOPS = (CHOP1 |||  CHOP2) ||| CHOP3

DINING = PHIS
[|{pick11,pick12,put11, put12, pick22, pick23,put22,put23, pick33, pick31,put31,
put33}|]
       CHOPS
```

CSP++ syntax:

```
       ...

SYS = PHIS
[|{pick11,pick12,put11, put12, pick22, pick23,put22,put23, pick33, pick31,put31,
put33}|]
       CHOPS
```

Timed Vending Machine

HORAE Syntax:

```
TVM = insert ->
      ((reqrelease -> release ->{3} TVM)
       [](coffee ->{3} dispatchcoffee -> TVM)
       [](tea ->{2} dispatchtea -> TVM))
       |\{10}(relese -> TVM)


TVM2 = insert ->
      ((reqrelease -> release ->{3} TVM2)
       [](coffee ->{3} dispatchcoffee -> TVM2)
       [](tea ->{2} dispatchtea -> Skip))


TVM3 = insert ->
      ((reqrelease -> release ->{3} TVM3)
       [](coffee ->{3} dispatchcoffee -> TVM3)
       [](tea ->{2} dispatchtea -> TVM3))
       |\{10}(relese -> Skip)

User = insert -> coffee -> tea -> Stop

Sys = TVM [|{insert, coffee, tea}|] User
```

CSP++ Syntax:

```
TVM = insert -> ((makechoice ->
((reqrelease -> release -3-> TVM) []
(coffee -3-> dispatchcoffee -> TVM) []
(tea -2-> dispatchtea -> TVM)))[10>
(release -> TVM))

TVM2 = insert ->
((reqrelease -> release -3-> TVM2)[]
 (coffee -3-> dispatchcoffee -> TVM2)[]
 (tea -2-> dispatchtea -> SKIP))

TVM3 = insert -> ((makechoice ->
((reqrelease -> release -3-> TVM3) []
(coffee -3-> dispatchcoffee -> TVM3) []
(tea -2-> dispatchtea -> TVM3)))[10>
(release -> SKIP))

--USER = insert -> SKIP
--USER = insert -> makechoice -> coffee -> SKIP
--USER = insert -> makechoice -> tea -> SKIP
```

```
--USER = insert -> makechoice -> reqrelease -> SKIP
--USER = insert -> SKIP
--USER = insert -> coffee -> SKIP
--USER = insert -> tea -> SKIP
--USER = insert -> reqrelease -> SKIP
--SYS = TVM [|{insert, coffee, tea, reqrelease, makechoice}|] USER
--SYS = TVM2 [|{insert, coffee, tea, reqrelease, makechoice}|] USER
SYS = TVM3 [|{insert, coffee, tea, reqrelease, makechoice}|] USER
```

Timed Railroad Crossing

HORAE Syntax:

```
Train = trainnear -> nearind ->{300} entercrossing ->{20} leavecrossing -
> outind -> Train

Controller = (nearind ->{1} downcommand -> confirm -> Controller) []
(outind ->{1} upcommand -> confirm -> Controller)

Gate =  (downcommand ->{100} down -> confirm -> Gate) [] (upcommand ->{100}
up -> confirm -> Gate)

Crossing = Controller || Gate

System = Train || Crossing
```

CSP++ Syntax:

```
TRAIN = trainnear -> nearind -300-> entercrossing -20->
            leavecrossing -> outind -> TRAIN

CONTROLLER = (nearind -1-> downcommand -> confirm -> CONTROLLER)
            [] (outind -1-> upcommand -> confirm -> CONTROLLER)

GATE =  (downcommand -100-> down -> confirm -> GATE) []
            (upcommand -100-> up -> confirm -> GATE)

CROSSING = CONTROLLER [|{|downcommand, upcommand, confirm|}|] GATE

SYS = TRAIN [|{|nearind, outind|}|] CROSSING
```

Appendix B: CSP++ Time Units

Currently CSP++ supports milliseconds, seconds, and minutes as specification time units for synthesis purposes. The time units can be specified by using the `pragma` key word. `pragma` is a part of FDR2 and ProBe syntax used to invoke special commands that are not part of the CSPm specification per se. Since any pragma that is unrecognized is simply ignored during the verification process, it appeared to be very useful for our needs. Below is an example of a simple CSPm specification that uses this pragma:

```
pragma timeunit m
channel a, b, c
SYS = a -5-> b -2-> c -> SKIP
```

`pragma timeunit` supports `ms` for milliseconds, `s` for seconds and `m` for minutes.

Note that the ability of the underlying scheduler to accurately delay small amounts of milliseconds depends on the timing arrangements in the CPU and operating system. Without the pragma, time units default to seconds.

Say a given specification calls for minutes as time units. However, during routine testing of the generated code it may take a long time to step though the whole program. To solve the problem, CSP++ also supports command line flags, `-ms` for milliseconds, `-s` for seconds, and `-m` for minutes, that take precedence over the specification time units. For example, say the above specification was synthesized with CSP++ and the execution code was contained in file called `test`. Running `./test -s` would cause the program to be executed with seconds for time units. Note, that command line flags do not change the specification permanently, and are simply used to speed up the debugging or routine simulation.

CSP specification and command line time units apply to the whole specification. A timeout, for example, cannot have different time units than timed prefix in the same specification. However, if the need for different time units arises, a programmer can use the smallest time unit as default, while multiplying larger time units accordingly. For example, if the specification calls for seconds and minutes, consider this:

```
pragma timeunit s
channel a, b, c, d
P = a -5-> b -> SKIP
Q = c -> SKIP [120> T
T = d -> SKIP
SYS = P ||| Q
```

Timed prefix in process `P` will be executed as 5 seconds, however, timed timeout in process `Q` will be executed as 120 seconds, which corresponds to 2 minutes.

When cspt would generate C++ code for the above specification, the statement

`int timeunit = 1000`;

will be included, which sets the time granularity to 1000 milliseconds or one second. Each time variable supplied as an argument to timed CSP++ functions will be multiplied by `timeunit` to set time units accordingly.

Appendix C: HORAE supported operators

Operator	CSP syntax	HORAE .tcsp syntax	CSP++ syntax
Delay	$a \xrightarrow{d} b$	a->{d} b	a -d-> b
Untimed Timeout	$Q1 \triangleright Q2$	Not supported	Q1 [> Q2
Timed Timeout	$Q1 \overset{d}{\triangleright} Q2$	Q1 |\{d} Q2	Q1 [d> Q2
Interrupt	$P \Delta Q$	P/\Q	Not supported[a]
Untimed Event Interrupt	$Q1 \Delta a \rightarrow Q2$	int(Q1, a, Q2)	Q1 /\ a -> Q2
Timed Interrupt	$Q1 \Delta_d Q2$	tint(Q1, d, Q2)	Q1 /d\ Q2
Deadlock	STOP	STOP	STOP
Successful termination	SKIP	SKIP	SKIP
Sequential Composition	P;Q	P;Q	P;Q
Interleaving	$P \parallel\mid Q$	P | | | Q	P | | | Q
Prefixing	$a \rightarrow P$	a -> P	a -> P
External (deterministic) choice	$P \,\square\, Q$	P [] Q	P [] Q
Internal (non-deterministic) choice	$P \sqcap Q$	P |~| Q	Not supported

Operator	CSP syntax	HORAE .tcsp syntax	CSP++ syntax
Wait	`Wait d`	WAIT(d)	Not supported
Timed Event prefix	`a@u`	a@u	Not supported
RUN	$\mathtt{RUN}_{\mathtt{A}}$	RUN(A)	Not supported
CHAOS	$\mathtt{CHAOS}_{\mathtt{A}}$	CHAOS(A)	Not supported
Channel Input	`c?m:T`[b]	c?m:T	c?m:T
Channel Output	`c!v`	c!v	c!v
Recursion	`N=P`[c]	N=P	N=P
Mutual Recursion	$\mathtt{N}_{\mathtt{i}} = \mathtt{P}_{\mathtt{i}}$[d]	N=P	N=P
Event Hiding	`P\A`[e]	P\A	P\A
Forward Rename	f(P)	P[[a <- b]]	P[[a <- b]]
Prefix Choice	$\mathtt{x:A} \rightarrow \mathtt{P(x)}$	x:A@a -> P(x)	Not supported

a. operators designated as “Not Supported” in the rightmost column do not limit CSP++. These operators can either be replaced by the combination of others or they may not be useful in software synthesis.

b. T defines the channel type.

c. N is the process name, and P is its body which may consist of named events.

Example: N = a -> b -> N. Process N performs events ‘a’ and ‘b’ in a loop.

d. N and P are process names. Example: ON = turn_off -> OFF; OFF = turn_on -> ON

e. P is the process name and A is the set of events removed from P’s interface and made internal to the process.

Appendix D: VAC CSP Specification and User-coded Functions

Timed CSP Specification:

```
channel forward, backward, left, right, done
channel manual, turn_on, turn_off, pickup
channel aforward, abackward, aleft, aright, adone

channel L_forward, L_backward, L_stop
channel R_forward, R_backward, R_stop
channel F_forward, F_backward, F_turn, F_stop

channel stopping_all_moving_parts, low_battery, going_to_base, good_bye


MOVEMENT_CONTROL = (aforward -> L_forward -> R_forward -> F_forward -1->
MOVEMENT_CONTROL) []
            (abackward -> L_backward -> R_backward -> F_backward -1->
MOVEMENT_CONTROL) []
            (aleft -> L_backward -> R_forward -> F_turn -1->
MOVEMENT_CONTROL) []
            (aright -> L_forward -> R_backward -> F_turn  -1->
MOVEMENT_CONTROL) []
            (astop -> L_stop -> R_stop -> F_stop -> MOVEMENT_CONTROL)[]
            (adone -1-> SKIP)


REMOTE_CONTROL =
(forward -> L_forward -> R_forward -> F_forward -1-> REMOTE_CONTROL)
                        []
(backward -> L_backward -> R_backward -> F_backward -1-> REMOTE_CONTROL)
                        []
(left   -> L_backward -> R_forward -> F_turn  -1-> REMOTE_CONTROL)
                        []
(right  -> L_forward -> R_backward -> F_turn  -1-> REMOTE_CONTROL)
                        []
(done  -1-> SKIP)


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

```
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
RUNNING = WHICHOPMODE /\ pickup -> EMERGENCY_STOP
WHICHOPMODE = (manual -> REMOTE_CONTROL) [> ((turn_off -> ROBOT(0)) [7>
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

--ROOM = aforward -1-> aleft -1-> aforward -1-> aright -1-> abackward -1-
> ROOM
--DIRT = dust -1-> DIRT


USER = turn_on -10-> pickup -> putdown -> SKIP
--USER = turn_on -> pickup -> SKIP
--USER = turn_on -> manual -> SKIP
--USER = turn_on -> manual -> forward -> done -> SKIP
--USER = turn_on -> manual -> forward -1-> left -> right -1-> pickup ->
SKIP
--USER = SKIP
--USER = turn_on -> turn_off -> SKIP
--USER = turn_on -> SKIP

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

ucfs.cc

```
#include "Lit.h"
#include "Action.h"

using namespace ucsp;

void turn_on_atomic( ActionType t, ActionRef* a, Var* v, Lit* l )
{
      cout << "\n-- VAC is powered on -- \n";
}

void turn_off_atomic( ActionType t, ActionRef* a, Var* v, Lit* l )
{
      cout << "\n-- VAC is powered off -- \n";
}

void pickup_atomic( ActionType t, ActionRef* a, Var* v, Lit* l )
{
      cout << "\n-- SENSORS: VAC has been picked up -- \n";
}

void put_down_atomic( ActionType t, ActionRef* a, Var* v, Lit* l )
{
      cout << "\n-- SENSORS: VAC has been put down -- \n";
}

void low_battery_atomic( ActionType t, ActionRef* a, Var* v, Lit* l )
{
      cout << "\n-- SENSORS: VAC's battery is low -- \n";
}

void clean_atomic( ActionType t, ActionRef* a, Var* v, Lit* l )
{
      cout << "\n-- Cleanind drum on. Air sucktion on. Cleaning... -- \n";
}

void stopping_all_moving_parts_atomic( ActionType t, ActionRef* a, Var*
v, Lit* l )
{
      cout << "\n-- EMERGENCY STOP! Stopping wheels! Stopping cleaning
drum! Stopping air sucktion! -- \n";
}

void L_forward_atomic( ActionType t, ActionRef* a, Var* v, Lit* l )
{
      cout << "\n-- LEFT wheel is rotating forward -- \n";
}

void L_backward_atomic( ActionType t, ActionRef* a, Var* v, Lit* l )
{
      cout << "\n-- LEFT wheel is rotating backward -- \n";
}

void L_stop_atomic( ActionType t, ActionRef* a, Var* v, Lit* l )
{
      cout << "\n-- LEFT wheel is stopped -- \n";
}

void R_forward_atomic( ActionType t, ActionRef* a, Var* v, Lit* l )
{
      cout << "\n-- RIGHT wheel is rotating forward -- \n";
}

void R_backward_atomic( ActionType t, ActionRef* a, Var* v, Lit* l )
{
      cout << "\n-- RIGHT wheel is rotating backward -- \n";
}

void R_stop_atomic( ActionType t, ActionRef* a, Var* v, Lit* l )
{
      cout << "\n-- RIGHT wheel is stopped -- \n";
}

void F_forward_atomic( ActionType t, ActionRef* a, Var* v, Lit* l )
{
      cout << "\n-- FRONT wheel is rotating forward -- \n";
}

void F_backward_atomic( ActionType t, ActionRef* a, Var* v, Lit* l )
{
      cout << "\n-- FRONT wheel is rotating backward -- \n";
}

void F_stop_atomic( ActionType t, ActionRef* a, Var* v, Lit* l )
{
      cout << "\n-- FRONT wheel is stopped -- \n";
}

void F_turn_atomic( ActionType t, ActionRef* a, Var* v, Lit* l )
{
      cout << "\n-- turning the FRONT wheel -- \n";
}
```