Supporting Selective Formalism in CSP++ with Process-Specific Storage

by

Alicia Gumtie

A Thesis
presented to
The University of Guelph

In partial fulfilment of requirements
for the degree of
Master of Science
in
Computing and Information Science

Guelph, Ontario, Canada

© Alicia Gumtie, September, 2012
Supporting Selective Formalism in CSP++ with Process-Specific Storage

Alicia Gumtie
University of Guelph, 2012

Advisor:
Dr. W. B. Gardner

Communicating Sequential Processes (CSP) is a formal language whose primary purpose is to model and verify concurrent systems. The CSP++ toolset was created to embody the concept of selective formalism by making machine-readable CSPm specifications both executable (through the automatic synthesis of C++ source) and extensible (by allowing the integration of C++ user-coded functions). However, these user-coded functions were limited by their inability to share data with each other, which meant that their application was constrained to solving simple problems in isolation. We extend CSP++ by providing user-coded functions in the same CSP process with safe access to a shared storage area, similar in concept and API to Pthreads’ thread-local storage, enabling cooperation between them and granting them the ability to undertake more complex tasks without breaking the formalism of the underlying specification. This feature’s utility is demonstrated in our line-following robot case study.
I would like to express my gratitude to the people who helped me along the way:

- My supervisor, Dr. Bill Gardner, who supplied guidance, patience, and an endless stream of puns;
- My parents, who always pushed me to excel in anything I did;
- My friends, who supported me and commiserated with me over academia and life;
- My husband, who did all of the above.
# Contents

Acknowledgments .................................................. i
Table of Contents .................................................. ii
List of Tables ..................................................... vi
List of Figures ...................................................... vii

1 Introduction ...................................................... 1
   1.1 Background .................................................. 2
   1.2 Problem Definition and Motivation ........................ 3
   1.3 Research Approach and Contributions ....................... 6
   1.4 Thesis Outline ............................................. 7

2 CSP and Associated Tools ...................................... 9
   2.1 CSP ......................................................... 9
      2.1.1 Processes and Events ................................. 10
      2.1.2 Concurrency and Communication .................... 12
      2.1.3 Control Flow ........................................ 14
      2.1.4 Hiding and Renaming ................................ 16
      2.1.5 Timed CSP ........................................... 17
      2.1.6 Traces ............................................... 21
   2.2 Verification Tools .......................................... 22
3 Related Technologies

3.1 Languages and Compilers ........................................... 25
  3.1.1 LOTOS .......................................................... 25
  3.1.2 occam and occam-pi ........................................... 25
  3.1.3 KRoC and NOCC ............................................... 26
3.2 Language Extensions ................................................ 26
  3.2.1 JCSP ............................................................. 27
  3.2.2 C++CSP2 ........................................................ 28
  3.2.3 PyCSP ............................................................ 28
  3.2.4 CSP.NET ....................................................... 29
  3.2.5 CHP ............................................................... 30
  3.2.6 CTC++ ........................................................... 30
3.3 Code Synthesizers .................................................... 30
  3.3.1 CCSP ............................................................. 31
  3.3.2 CSP to JCSP .................................................... 31
  3.3.3 CSPtoHC (CSP to Handel-C) .................................. 32
  3.3.4 gCSP ............................................................ 32
3.4 Tools Using CSP Internally ......................................... 33
  3.4.1 ASD .............................................................. 33
  3.4.2 R2D2C ........................................................... 34

4 CSP++ .......................................................................... 35
4.1 Overview ............................................................... 35
4.2 History ................................................................. 36
  4.2.1 CSP Dialect Supported ......................................... 36
  4.2.2 Platforms Supported ............................................ 37
  4.2.3 Micro CSP++ .................................................... 38
  4.2.4 Timed CSP ....................................................... 39
<table>
<thead>
<tr>
<th>Section</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>8.2.1 General CSP++ Improvements</td>
<td>103</td>
</tr>
<tr>
<td>8.2.2 Process-Specific Storage Improvements</td>
<td>105</td>
</tr>
<tr>
<td>Bibliography</td>
<td>107</td>
</tr>
<tr>
<td>A Source Files for PID-Controlled LEGO Line Follower</td>
<td>115</td>
</tr>
<tr>
<td>A.1 C++ User-Coded Functions</td>
<td>116</td>
</tr>
<tr>
<td>A.2 Makefile</td>
<td>125</td>
</tr>
<tr>
<td>B Guide to New Features in CSP++ V5.1</td>
<td>127</td>
</tr>
<tr>
<td>B.1 New Translator Features</td>
<td>127</td>
</tr>
<tr>
<td>B.2 Process-Specific Storage Usage Guide</td>
<td>128</td>
</tr>
</tbody>
</table>
List of Tables

2.1 Comparison of Timed CSP notations ............................ 20

5.1 Proposed storage behaviour ....................................... 57
5.2 Comparison of process-specific storage and Pthreads'/Pth's TLS 59
5.3 Agent branches values and corresponding process structures . 61

6.1 User-coded functions in robot case study ......................... 70

7.1 Average CSP++ program run times (s), with and without storage 95
7.2 Estimated CSP++ program memory usage (KB), with and without storage ................................................. 96
7.3 Average CSP++ program run times (s), with dynamic and static linking ......................................................... 100
7.4 CSP++ executable sizes (KB), with dynamic and static linking . 100
List of Figures

4.1 CSP++ design flow ........................................ 40
4.2 CSP++ class diagram ......................................... 42

6.1 LEGO line-following robot .................................. 66
6.2 System diagram ................................................ 67
6.3 Sensor array positions for line detection cases .............. 68
6.4 FDR2 deadlock test results .................................. 79
6.5 FDR2 livelock test results .................................. 79
6.6 FDR2 determinism test results ............................... 80
6.7 FDR2 trace 1 refinement check results ....................... 81
6.8 FDR2 trace 2 refinement check results ....................... 83
6.9 FDR2 trace 3 refinement check results ....................... 90
Chapter 1

Introduction

Communicating Sequential Processes (CSP) [Hoa78] is a formal language whose primary purpose is to model and verify concurrent systems. The CSP++ toolset [W. 09] makes machine-readable CSP specifications both executable (through the automatic synthesis of C++ source) and extensible (by allowing the integration of C++ user-coded functions).

This research enhances the CSP++ toolset with the ability to attach local storage to a CSP process, thereby providing a way for CSP events to communicate within a process in addition to the traditional communication between processes. Prior to this addition, there were certain classes of problems that CSP++ could not be used to solve, as they contained complex functionality that was difficult or impossible to express in either a CSP specification or C++ user-coded functions. The addition of process-specific storage allows system designers to implement this functionality inside cooperating C++ user-coded functions, widening the applicability of CSP++.

In this chapter, we first describe the advantages that formal methods and automatic code synthesis can furnish a software developer. Next, we introduce CSP++ and its foundational concept of selective formalism. We then
outline the problem of user-coded functions’ limited utility and supply reasons for tackling it. Finally, we give a synopsis of our research approach and contributions, and provide an overview of the remainder of the thesis.

1.1 Background

Software is ubiquitous; it controls vehicles, appliances, communication devices, buildings, robots — the list goes on. However, due to ad-hoc software development practices, the lack of rigorous testing, incomplete or ambiguous requirements, and a multitude of other factors, not all of this software is well-formed or accurate. Poorly-written software can have a catastrophic effect on people’s safety and well-being, as can be seen in the LASCAD fiasco of the 1990s [FD96]: London’s then-new computerized ambulance dispatch service erased emergency calls, misallocated ambulances, jammed call queues with inaccurate automatic alerts, and so mangled the emergency system that it was taken offline 35 hours after its deployment. Up to 46 emergency patients may have died prematurely in that time for lack of emergency service [Dal99]. During the subsequent investigation it was discovered that the developers of the dispatch software had no prior experience in creating a time-critical and safety-critical application.

This is the type of scenario that formal methods can help avoid — by allowing software developers to reason about their systems mathematically, formal methods can help produce provably correct implementations. However, the use of formal methods does not guarantee a correct implementation, as manual translation of a formal specification into source code may introduce new errors in the system, and a formal specification becomes a more inaccurate representation of its implemented system as the system un-
dgoes modifications.

These problems can be alleviated through automated code synthesis; the formal specification and the generated code will be always representative of the same system, correct by construction, and remain synchronized through any changes, given that the translation process employed by the synthesis tool is correct.

Regardless of the advantages that formal methods offer, they are often seen as difficult and costly when compared to traditional software development practices. In particular, specification of an entire system is generally a daunting or infeasible task. In response to this, Gardner’s concept of selective formalism [Gar03] provides a hybrid approach: rather than requiring a system designer to formally specify an entire system, one only need apply formal methods where they will be most useful — in modelling concurrency in the system’s backbone. The remainder of the system’s functionality, such as data processing and I/O, can be implemented directly in a traditional programming language, without prior formal specification or verification.

The CSP++ toolset [Gar00], discussed in Chapter 4, embodies the concept of selective formalism. It allows the synthesis of C++ system control code from a specification written in the CSP formal language, and the extension of that code’s functionality via user-coded C++ functions (UCFs) attached to CSP events.

1.2 Problem Definition and Motivation

One of selective formalism’s main goals is to be able to replace some or all of a CSP process with C++ UCFs after verification and simulation. Since CSP is designed to be a formal modelling language, not a full-fledged program-
ming language, the best candidates for replacement are pieces of functionality that either do not require formal verification (such as calculations) or are not well suited to being expressed in CSP (such as data structures). Since a programming language gives more freedom of expression than a formal language when implementing rather than simulating, this would facilitate the creation of complex systems not fully or easily expressible in CSP, with a verifiably-correct formal control spine.

Prior to this point, this goal had not been been fully realized in CSP++, even though UCFs existed. We were able to replace individual events and channels effectively; however, the resulting UCFs were forced to work in isolation. To replace a set of them to accomplish some task would require UCFs to maintain some kind of state information between them, but CSP++ did not provide a way for them do that safely, without breaking the underlying formalism. This problem is further discussed in Chapter 5, but three use cases of particular note are:

- A single operation that needs input from the control backbone and produces output to be returned to it. This requires 2 UCFs, one to take input and create the output, and another to transmit the output to the backbone. However, the second UCF cannot transmit data it has no knowledge of — the first UCF must communicate its result to the second. For instance, a vending machine might want to calculate how much change to return to a customer. The first UCF takes the value of the inserted money, subtracts the item price, and communicates the result to the second UCF, which sends the result out to the formally-modelled code for later use.

- A series of operations that can hold their data internally, outside of the CSP specification, because it does not affect the system’s control flow.
All UCFs involved in these operations must have access to this data. A
good example of this scenario is given in Chapter 6 — we perform a
series of calculations, one per UCF, each of which uses the result of a
previous one in its own algorithm. Each calculation's result must be
made available to subsequent UCFs.

- A series of operations where the CSP backbone accesses and updates
  persistent state. State can only be preserved in process parameters if
  it is representable by integers alone; more complex datatypes must be
  externally represented. The ATM case study [Dox05] handles this by
  storing its state in an SQL database, but this induces more interprocess
  communication and synchronization than is necessary for the base sys-
  tem to operate.

It is possible to emulate true intraprocess communication by breaking a
single process into multiple ones and passing parameters between them, but
this approach has some drawbacks:

- Process semantics are destroyed. A useful aspect of CSP is the ability
to mirror a real-world process with a CSP process. If the CSP process is
separated into multiple processes, the parallels between the model and
the actual system are less clear.

- The CSP specification is cluttered, making it more difficult to under-
stand and modify.

- Reconfiguration of the specification may be necessary to guarantee that
the chain of events comprising the processes will be unbroken or unin-
terrupted, as it would be if they were residing in a single process. One
of the partial processes may end and another unrelated process in the
model may begin before the next partial process commences.
- The generated C++ code is inflated and obfuscated.

Ideally, we would avoid these problems by using a set of UCFs that would work together to implement a complex bit of functionality by individually manipulating shared data. Of course, this manner of cooperation between UCFs in different processes would constitute a violation of the formalism, since all interprocess communication should be modeled in the CSP specification. Cooperation between UCFs in the same process, however, is not a risk. This meant that a set of cooperating UCFs had to be composed of UCFs attached to the same CSP process, and all data shared between them had to be strictly local to that process — thus the creation of process-specific storage.

It is the thesis statement of this research that enhancing CSP++ to provide process-specific storage to UCFs will allow implementation of a wider range of algorithms and systems, a subset of which are represented by the use cases discussed above.

1.3 Research Approach and Contributions

There were several steps involved in the addition of process-specific storage to CSP++:

The greatest challenge lay in defining the ideal behaviour for the storage feature. Much effort was devoted to the research and consideration of CSP process identity and control flow, as these factors would affect storage scope and data propagation. In addition, we had to ensure that the introduction of storage would not have presented any chances for violation of a CSP specification's formalism. More detail on this can be found in Chapter 5.

The implementation phase required additions and modifications to the
CSP++ object-oriented application framework. This necessitated more research, this time into the history and nuances of CSP++ itself. Our findings are summarized in Chapter 4. We also implemented other new features in the CSP++ translator (see Appendix B.1), namely support for the translation of CSP$m$ constants, as well as modulo and boolean operators; these were intended for use in our case study, though ultimately only the translation of constants was employed. After the process-specific storage feature was created, we devised a set of regression tests to build confidence in the correctness of its implementation.

Once complete, we needed to demonstrate the utility of storage through a case study. This afforded us the opportunity to showcase CSP++ user-coded functions far more extensively than had been previously accomplished. Our line follower, with its Proportional-Integral-Derivative (PID) controller, was the first CSP++ case study to be successfully implemented in the real world, and not just as a simulation. The case study is discussed in Chapter 6.

### 1.4 Thesis Outline

The remainder of this thesis is organized into several chapters. Chapter 2 provides an introduction to the CSP formal notation and the concepts it embodies. Chapter 3 reviews works related to CSP++ in goals and methodologies. Chapter 4 outlines the structure and functionality of the CSP++ tools. Chapter 5 discusses the addition of process-specific storage to CSP++. Chapter 6 presents the PID-controlled line follower case study, including the CSP specification. Source code for the case study’s UCFs can be found in Appendix A. Chapter 7 examines the performance of CSP++ before and after the storage modifications. Chapter 8 suggests some directions for future work on CSP++
and gives conclusions on this research. Finally, Appendix B contains a guide to the features introduced in this work.
Chapter 2

CSP and Associated Tools

This chapter provides a brief introduction to CSP and the verification tools used with it. Section 2.1 outlines CSP’s uses and mechanics with sample specifications written in machine-readable CSPm, and Section 2.2 gives an overview of commercial and academic CSP verification tools available for use.

2.1 CSP

CSP is a formal language created by Hoare in 1978 [Hoa78]. Its primary purpose is to model concurrent systems as sets of processes, and in doing so to eliminate, or at least make detectable, the mistakes and pitfalls that often occur when reasoning about and implementing concurrency. In particular, concurrent systems are prone to errors that may not occur in sequential systems, such as deadlock, livelock, and nondeterminism.

Practically, there are four types of model that CSP is used for [Gar05]:

- The *functional model* expresses system behaviour as CSP constructs — processes and events.
- The *environmental model* simulates external entities that are part of the system’s intended environment. If synchronized with the functional model, the functional model can be simulated.

- The *constraint model* defines the “safety” aspects of a functional model by constraining its possible execution paths with more processes. It is used to verify that the functional model meets its safety requirements.

- The *implementation model* is a less abstract, more detailed representation of a system than the functional model. This model type provides a starting point for code synthesis.

To demonstrate the features of CSP, we will make use of the classic vending machine scenario and modify it as we progress. In accordance with CSP convention, process names are in uppercase, while event and channel names are in lowercase.

### 2.1.1 Processes and Events

At the core of CSP lies the concept of the process. A real-world system is composed of many interacting components; a basic CSP process is meant to be an abstraction of a single one. Note that a CSP process, as a concept, is distinct from and unrelated to an operating system process. All mentions of “processes” in this work refer to CSP processes.

Our hypothetical vending machine is highly abstracted – it will be represented by only one process. In reality, it could be modeled with far more: one for the change calculating machine, one for the mechanical dispenser, one for the temperature control unit, and so on. The level of detail used is left to the specification author’s discretion, but will generally depend on the purpose for which the specification is intended.
Let us assume the simplest of vending machine setups: the machine accepts a coin and dispenses a drink, infinitely. In the following specification, \texttt{VM} is the CSP process that comprises the functional model of our machine.

\begin{verbatim}
VM = coin -> drink -> VM
\end{verbatim}

A process is composed of events. These events represent atomic points or actions of interest within the real-world system component; in our example, the coin being inserted (\texttt{coin}) and the drink being dispensed (\texttt{drink}) are the interesting bits that we want to model. The prefix operator (\texttt{->}) represents the transition from one event to the next. A process has a list of events it can perform; this list is the process' \textit{alphabet}.

Every process in CSP must terminate in a process (called \textit{chaining}). Note that the \texttt{VM} process ends by invoking itself; this allows it to continue recursively forever. For actual termination, CSP provides two processes, \texttt{SKIP} and \texttt{STOP}. \texttt{SKIP} represents termination under normal conditions; \texttt{STOP} represents termination under abnormal conditions. After executing either of these, a process can do nothing else.

A CSP process may be parameterized; this allows the passing of data upon process termination, and, consequently, the creation of finite loops. For instance, if our vending machine could only dispense three drinks, it might look like this:

\begin{verbatim}
VM(0) = SKIP
VM(i) = coin -> drink -> VM(i-1)
SYS = VM(3)
\end{verbatim}

The \texttt{SYS} process invokes the \texttt{VM} process with a parameter of 3. The vending machine accepts a coin and dispenses a drink three times, decrementing
its parameter each time. When the parameter equals zero, it terminates.

2.1.2 Concurrency and Communication

The real power of CSP lies in its ability to model the interactions between components. To do this, it provides two types of parallel process composition: synchronized and interleaved. Synchronized parallelism requires processes to synchronize on a subset of the common events in their alphabets. All processes synchronizing on an event must perform that event simultaneously; if some processes are not ready to perform that event, the others must wait. We can demonstrate this by introducing an environmental model in the form of a customer (CUST).

<table>
<thead>
<tr>
<th>VM = coin -&gt; drink -&gt; VM</th>
</tr>
</thead>
<tbody>
<tr>
<td>CUST = approach -&gt; coin -&gt; drink -&gt; retrieve -&gt; leave -&gt; SKIP</td>
</tr>
<tr>
<td>SYS = VM [{ coin, drink }</td>
</tr>
</tbody>
</table>

The SYS process models the interaction between the functional model and the environmental model; that is, the vending machine and its simulated customer. They synchronize with each other on the specified set of events, denoted by [{ coin, drink }]. Since VM and CUST are synchronizing on the coin and drink events, neither of them can perform those events independently. The vending machine must wait for the customer to approach before they can execute the coin deposit and drink dispensing together. Only after those two events have occurred can the customer take his drink and leave the machine.

The other type of parallelism is interleaved. In interleaved parallelism, processes execute completely independently of each other; no synchroniza-
tion occurs, even if the processes perform the same events. A good example of this would be a scenario with two vending machines.

\[
\begin{align*}
\text{VM1} &= \text{coin} \rightarrow \text{drink} \rightarrow \text{VM1} \\
\text{VM2} &= \text{coin} \rightarrow \text{drink} \rightarrow \text{VM2} \\
\text{SYS} &= \text{VM1} \mid \mid \text{VM2}
\end{align*}
\]

Even though both machines can execute \textit{coin} and \textit{drink}, each instance of those events is distinct and unrelated to any others. \textit{VM1} and \textit{VM2} do not have to wait for each other to be ready before those events can be performed.

Another crucial feature of CSP is its ability to model communication between processes. Communication is performed over unidirectional, process-to-process \textit{channels}. Each channel has a set of messages that can be transmitted over it, called its \textit{type}. Transmission on a channel is an event and may be treated as such; of particular note is the fact that message passing in this way is synchronizable. We can demonstrate this if we modify our vending machine to display the value of the coin inserted.

\[
\begin{align*}
\text{channel display} : &\{1, 5, 10, 25, 100, 200\} \\
\text{VM} &= \text{coin} \rightarrow \text{display}\text{!value} \rightarrow \text{drink} \rightarrow \text{VM} \\
\text{CUST} &= \text{approach} \rightarrow \text{coin} \rightarrow \text{display}\text{?value} \rightarrow \text{drink} \rightarrow \text{retrieve} \rightarrow \text{leave} \rightarrow \text{SKIP} \\
\text{SYS} &= \text{VM} \mid [\mid\{\mid\text{coin, drink, display} \mid\}] \mid \mid \text{CUST}
\end{align*}
\]

In this example, line 1 declares a channel named \textit{display} and specifies its type (the set of integers shown). The vending machine outputs (\texttt{!}) the coin's value on the channel, while the customer receives that data as input (\texttt{?}) on the channel. The data transmitted will be one of the entries specified in the
display channel's type.

Compare this to the example where we first introduced the customer. There is a critical difference in the nature of the synchronizations involved. In the former example, synchronization occurs on a set of atomic events that do not involve channel communication (denoted \([\{\ldots\}])\). In this last example, synchronization occurs on the set closure of those events — that is, on the events themselves as well as on all instances of data communicated over the specified channels (denoted \([\{|\ldots\}|\] )\).

Note that this is the only method of communication between processes in CSP. No process can access another's data directly; all messages must be explicitly passed. This message-passing model addresses a weakness in the shared-memory model of concurrency; without globally accessible data, there is no need for complex and error-prone locks and critical sections.

### 2.1.3 Control Flow

A process is not necessarily tied to a single path of execution. CSP includes features that make multiple execution paths possible. These are conditional statements, external choice, internal choice, and sequential composition.

The *conditional* construct (*if-then-else*) allows a process to proceed based on the evaluation of a given expression. Supported types are inequalities, arithmetic expressions and boolean expressions. We can demonstrate this via modification of our customer’s behaviour; if the value of his money on hand is enough to buy a drink, he will, otherwise he will leave empty-handed.

```
VM = coin -> drink -> VM
CUST(c) = approach -> if (c >= 100) then coin -> drink ->
    retrieve -> leave -> SKIP
```
Since our customer only has half the amount required to buy a drink, he is forced to go without.

When a process can offer to perform more than one event, it can present an \textit{external choice} (\{\}) to allow the environment to determine its path of execution. If we enable our vending machine to return the deposited coin as well as serve drinks, the customer can decide which action it takes.

\begin{verbatim}
SYS = VM [\{ coin, drink \}] CUST(50)

else leave -> SKIP

SYS = VM [\{|{ coin, drink |}\}] CUST(50)
\end{verbatim}

After a coin is inserted, the vending machine is offering to either dispense a drink or give a refund; the path it takes is determined by the customer. Since a refund event is available and a drink event is not, VM follows the refund path.

In contrast, \textit{internal choice} (\{\}) represents a decision that the system makes internally, which the environment cannot influence. For instance, our vending machine might decide to dole out a free drink once in a while:

\begin{verbatim}
VM = coin -> ( drink -> VM
               [] refund -> VM )

CUST = approach -> coin -> refund -> leave -> SKIP

SYS = VM [\{|{ coin, drink, refund }\}] CUST
\end{verbatim}

The vending machine itself knows how to make the decision, but to the environment, that choice is a black box; there is no insight into how it is
made and no way to interface with it. Because of this, it is a nondeterministic operator, and therefore of limited utility when trying to specify how a system should behave (for code synthesis) rather than how it can behave (for abstract modelling).

Sequential composition (;) allows control to pass from a terminated process to another one. The first component process executes until it terminates, upon which the second component process begins its own execution. The entire composition is not considered finished until its last component process has terminated. We can use this to model a chain of customers visiting the vending machine:

```
VM = coin -> drink -> VM
CUST1 = approach -> coin -> drink -> retrieve -> leave -> SKIP
CUST2 = approach -> coin -> drink -> retrieve -> leave -> SKIP
CUSTS = CUST1 ; CUST2
SYS = VM [{ coin, drink }||] CUSTS
```

VM performs coin and drink with CUST1 first, and then again immediately afterward with CUST2.

### 2.1.4 Hiding and Renaming

CSP offers some convenient abstraction features. Event hiding (\) is one; it allows a given set of events to be obscured from the environment and executed “behind the scenes”. This is useful when trying to declutter the interface between processes and when performing events that the environment should not be aware of. If we assume the customer always inserts at minimum sufficient change to make a purchase, then we can modify the vending machine
to calculate the difference and make the necessary refund.

```
VM = ( coin -> drink -> calc_change -> refund -> VM ) \ { calc_change }
CUST = approach -> coin -> drink -> refund -> leave -> SKIP
SYS = VM [||{ coin, drink }||] CUST
```

Because calc_change is an internal event, we can encapsulate it inside its related process and hide it from the rest of the system. The environment does not see it occur.

Event renaming ([[ a <- b ]]) allows a process to be cloned, with a subset of its events renamed. This promotes code reuse and is convenient when two processes have only slightly different specifications. To demonstrate, we can introduce a new vending machine that sells snacks instead of drinks:

```
VM_DRINK = coin -> drink -> VM_DRINK
VM_SNACK = VM_DRINK [[ drink <- snack ]]
```

This effectively replaces the drink event in VM_DRINK with the snack event in VM_SNACK.

### 2.1.5 Timed CSP

Timed CSP is an extension of Hoare’s CSP by Schneider [Sch00] developed in order to describe systems with time-sensitive behaviour. The CSP specifications to this point have been written in the CSPm dialect, which is used in the commercial verification tools created by Formal Systems (Europe) Ltd. and which CSP++ supports. These tools are some of the very few that embody the particular semantics of Roscoe’s and Schneider’s interpretations of CSP, which CSP++ uses to define correct behaviour. However, CSPm does not
include Timed CSP operators. Since one of CSP++’s intended uses is the specification and implementation of soft real-time systems, it was necessary to provide a way to describe timing in a CSP specification. Timed CSP operators were implemented by Solovyov [Sol08] as part of his masters degree.

Before discussing timed CSP operators, we must first discuss their traditional untimed counterparts. The first, prefix (\(\rightarrow\)), we have already seen. It transitions from one event to another instantaneously.

The second is timeout (\(\triangleright\)). The name is a bit misleading; untimed timeout, like all untimed operators, is triggered by events occurring, not time elapsing.

Untimed timeout implies that the process is in a state in which it may or may not be offering to perform the given event. Whether or not this event is actually available is unknown to the rest of the system, and so the operator is nondeterministic. This is not useful from a code synthesis perspective, so the CSP++ implementation of untimed timeout is similar to a polling action; we try an event and if it succeeds, the process continues as the left-hand operand; otherwise, it continues as the right-hand operand. This deterministic operation produces the same traces as the nondeterministic timeout.

As an example, the customer has two courses of action based on whether the user-unfriendly VM can dispense a drink:

\[
\begin{align*}
1. & \text{VM} = \text{coin} \rightarrow \text{drink} \rightarrow \text{VM} \\
2. & \text{CUST} = \text{approach} \rightarrow \text{coin} \rightarrow \text{drink} \rightarrow \text{DISPENSED} \rightarrow \text{SOLD_OUT} \\
3. & \text{DISPENSED} = \text{retrieve} \rightarrow \text{leave} \rightarrow \text{SKIP} \\
4. & \text{SOLD_OUT} = \text{kick} \rightarrow \text{leave} \rightarrow \text{SKIP} \\
5. & \text{SYS} = \text{VM} [\{\text{coin}, \text{drink}\}] \upharpoonright \text{CUST}
\end{align*}
\]
The third is *interrupt* (\`). A potentially interrupting event is registered with a process. If the interrupting event occurs, the process immediately terminates and shifts control to another specified process. We can demonstrate this by preparing our vending machine for a loss of electricity (assuming it contains a small independent power supply):

```
VM = coin -> drink -> VM /\ outage -> OFF
OFF = shutdown -> SKIP

POWER = good -> POWER
       [] outage -> STOP

SYS = VM ||| POWER
```

As long as the flow of electricity to the vending machine is unbroken, it operates normally. If a power outage occurs, the machine ceases normal operation and initiates a clean shutdown.

The timed versions of these operators are similar; the difference is that they incorporate time delays into their operations instead of occurring based on an event. Delay duration is represented by an integer we will denote as \( t \). In CSP++, the default time unit is seconds, but time units of milliseconds, seconds, minutes, and hours can be selected via either a `pragma` line in the CSP specification or a switch (`-ms`, `-sec`, `-min`, `-hr`) during program execution.

*Timed prefix* (-t-) still represents a process’ transition from one event to another, but now there is a minimum delay of time \( t \) that must elapse between the events occurring.

```
VM = coin -5-> drink -> VM
```
In this example, after the coin event is executed, the process must wait at least 5 time units before executing the drink event.

Timed interrupt (/t\) will interrupt a process when the specified time has elapsed, not when any other event has occurred. As an example, the vending machine will shut down after fifteen time units, whether or not a transaction has taken place or is currently taking place:

\[
\begin{align*}
\text{VM} &= \text{coin} \rightarrow \text{drink} \rightarrow \text{VM} /15\ ackslash \ 	ext{OFF} \\
\text{OFF} &= \text{shutdown} \rightarrow \text{SKIP}
\end{align*}
\]

Timed timeout ([t>) specifies that the left-hand process has t time to begin, otherwise the right-hand process will begin instead. To contrast with a timed interrupt, here our vending machine will shut down after fifteen time units only if a transaction is not initiated within that time.

\[
\begin{align*}
\text{VM} &= \text{coin} \rightarrow \text{drink} \rightarrow \text{VM} [15> \ 	ext{OFF} \\
\text{OFF} &= \text{shutdown} \rightarrow \text{SKIP}
\end{align*}
\]

As previously stated, these machine-readable timed operators are not part of the CSPm dialect (and are therefore incompatible with the Formal Systems (Europe) tools). They were created by Solovyov [Sol08] as part of his masters thesis work on CSP++. Their basic shapes echo the machine-incompatible symbols used in Schneider’s notation. Table 2.1 compares the two notational styles.

**Table 2.1: Comparison of Timed CSP notations**

<table>
<thead>
<tr>
<th>Timed Operator</th>
<th>Schneider</th>
<th>CSP++</th>
</tr>
</thead>
<tbody>
<tr>
<td>prefix</td>
<td>( \rightarrow ) ( t )</td>
<td>( -t-&gt; )</td>
</tr>
<tr>
<td>interrupt</td>
<td>( \triangle t )</td>
<td>/t\</td>
</tr>
<tr>
<td>timeout</td>
<td>( \triangleright )</td>
<td>[t&gt;)</td>
</tr>
</tbody>
</table>
2.1.6 Traces

A process in CSP is defined by its trace; that is, the finite sequence of events that it is observed to have performed. A single trace does not necessarily represent a complete execution of the process, from start to finish; instead, it is a snapshot of the process’ execution up to a given point in time. A process’ set of traces, represented as $\text{TRACE(process)}$, is the set of all possible sequences of events that the process can perform. As a simple example, the set of traces for a vending machine that dispenses a single drink then breaks:

\begin{verbatim}
  VM = coin -> drink -> STOP

  TRACE(VM) = { <>, <coin>, <coin, drink> }
\end{verbatim}

Note the inclusion of the empty trace ($<>$). All processes’ sets of traces must include the empty trace, as it is possible that a process will do nothing.

Traces enable analysis and verification of process behaviour. Two useful process properties are trace equivalence and trace refinement. If two processes have the same set of traces, then the processes are trace equivalent. If one process’ set of traces is a subset of another process’ set of traces, then the first process is a refinement of the second. These properties allow verification of the fact that sets of traces do actually conform to a given high-level specification.

It is worth noting that hidden events do not appear in traces, and so do not affect trace equivalence or refinement.
2.2 Verification Tools

The main commercial verification tools for CSP specifications are the product of Formal Systems (Europe) Limited [fse10]. FSEL has created three tools to aid the CSP specification author:

- **checker** is a syntax checker. It does not perform any semantic checking.

- **ProBE** is a process behaviour explorer. It allows users to interactively navigate the state space of a specification and to examine the resulting traces.

- **FDR2** is a model checker. It demonstrates the presence of deadlocks, livelocks, and nondeterminism in a given specification by displaying the offending portions of the specification to the user. It can also determine whether one specification is a refinement of another.

ProBE and checker are both freely available, as is FDR for academic use. The use of FDR for commercial purposes requires a license.

All of these tools support the machine-readable dialect of CSP called CSPm. To ensure compatibility with the FSEL tools, CSP specifications written for use with CSP++ must also be in the CSPm dialect. Therefore, all CSP specifications in this document are written using CSPm, with the exception of timed operators.

None of the FSEL tools support Timed CSP operators. FDR2 version 2.94 introduces a new type of choice operator \( (P \ [\! +A\! ] \ Q) \) which excludes events in a specified process \( (A) \) from resolving the choice. This can presumably be used to allow synchronization with CSP’s global clock via the `tock` event. The concept of a global clock continually producing `tock` events is native to standard CSP, and is not used in Timed CSP.
A Timed CSP verifier called HORAE [DHSZ06, DZS09, hor10] has been under development at the National University of Singapore since 2006. It has been extended to also verify Timed Planning, a new formal language based on Timed CSP. Unfortunately, it is not yet available for use. PAT [LSD11, pat12], also from the National University of Singapore, is an extensible framework for model checking and simulation. It currently supports the specification of some Timed CSP features in its Real-Time System module using a homegrown CSP extension called CSP#, the syntax of which is incompatible with CSPm.
Chapter 3

Related Technologies

Projects that aim to integrate CSP into the usual development workflow generally take one of four forms:

- languages and compilers
- language-specific extensions that implement CSP structures
- code synthesizers
- tools using CSP internally

For the most part, the following projects’ goals differ from CSP++’s own. They focus on providing CSP-like constructs or the correctness that CSP verification can impart, but as CSP++ allows synthesis and customization of executable code from a formal specification of just the system backbone, they do not quite offer the same breadth of coverage of the software development process.
3.1 Languages and Compilers

Generally, these languages implement the features of formal concurrent languages (CSP and others) natively. Examples are LOTOS, occam, and occam-pi.

3.1.1 LOTOS

LOTOS [ISO89] is a concurrent programming language based on CSP. It was created by ISO in 1989 to be used in the formal specification of OSI layer protocols, but it may also be used to describe other distributed systems. LOTOS holds ISO standard number 8807. LOTOS does not provide any intrinsic method of formal verification, though Fernandez et al. [FGM+92] created a toolbox to verify LOTOS programs.

3.1.2 occam and occam-pi

Occam [SGS95] is a language that implements CSP's process-channel model. It is used for both software creation and hardware description. Occam was created by INMOS Ltd. (now SGS-Thompson Microelectronics Ltd.) in 1982, though they are no longer developing the language or its compiler. The latest (and possibly last) version of occam, 2.1, was released in 1995.

Occam supports the concept of time, but does not provide direct counterparts to Timed CSP operators; instead, it provides a timer construct which can be used in conjunction with channels, deterministic choice (“alternation” in the occam parlance), and other constructs implemented from the original CSP.

Occam-pi [WB05, wel10] is a variant of occam created at the University of Kent. It extends the original occam with Milner’s pi-calculus [MPW92], a younger (relative to CSP) member of the process algebra family. The most
notable feature introduced is mobility; data, channel ends, and processes can all be communicated over channels [WB05].

Occam-pi has been implemented on the LEGO Mindstorms. It runs on the Transterpreter virtual machine [JJ05, SJJ07].

### 3.1.3 KRoC and NOCC

KRoC [kro10], the Kent Retargetable occam Compiler, is a toolchain for occam and occam-pi development. The latest version of KRoC currently supports Intel/i386 based machines running Linux or FreeBSD. Older, binary-only versions are available for Solaris/SPARC, OSF/Alpha and Parsytec/PowerPC, but they lack occam-pi support.

KRoC allows CSPm specifications to be extracted from occam-pi code [BR09] to facilitate formal verification; this is the opposite of the CSP++ approach (early verification, before any compilable code is written or generated).

NOCC [Bar06], the New occam-pi Compiler, is an experimental occam-pi compiler also under development at Kent, meant to replace the compiler KRoC currently includes. It compiles machine-readable CSP (the NOCC team's own dialect, called MCSP [mcs10], which is incompatible with CSPm) to native code with the help of KRoC.

### 3.2 Language Extensions

These extensions provide supplementary concurrency models to the ones already present in traditional programming languages. A library implements core CSP constructs (processes, channels, choice, and so on) for a particular
language. Programmers can then integrate these constructs with their own code.

The general focus of these extensions is on the structural aspects of CSP, rather than the formal; as such, most of them do not provide any facilities for formal verification. Code synthesis and Timed CSP support is similarly limited.

The University of Kent spawned a good portion of these libraries; JCSP (for Java) [WBM+07, jcs10], C++CSP(2) (for C++) [Bro07, bro10], and CHP (for Haskell) [Bro08].

The other libraries are also European; CSP.NET (for the .NET platform) [LO06] and PyCSP [BVA07] are both projects of the University of Copenhagen, though PyCSP is a joint undertaking with the University of Tromsø. CTC++ (also for C++) [OB04] is developed at the University of Twente.

### 3.2.1 JCSP

JCSP has been in development at the University of Kent since 1997. The latest published version is JCSP 1.1; it is open-source under the LGPL. It can currently be used with Java versions greater than or equal to 1.1, though support for Java 1.1 and 1.2 may be dropped in the future in favour of using features (particularly the collection classes) introduced in Java’s newer iterations [WBM+07].

An enhanced version of JCSP, JCSP Network Edition, had a brief commercial stint under the spinoff company Quickstone Technologies Limited, but the company was shuttered and JCSP-NE’s features were integrated into the academic codebase [WBM+07].

JCSP differs from the other libraries in that a third-party formal verifier [Rum05] has been written for it. Rümmer et al. built upon an existing tool
(KeY [ABB+05]) for the verification of JavaCard (a threadless, simplified Java) programs. Their new tool only verifies programs that use strictly JCSP, not programs that circumvent it and use native Java functionality such as shared memory.

A stripped-down version of JCSP, JCSPre [KLP08], has been used to control a LEGO NXT robot. The robotics API and virtual machine are courtesy of the leJOS NXJ project [lej10].

JCSP channels and deterministic choice were formally proven to provide the same behaviour as their CSP counterparts in 2000 [WM00], but whether that analysis remains relevant to the current incarnation of JCSP is unclear.

### 3.2.2 C++CSP2

C++CSP2 is another product of the University of Kent. The most recent version was released in January 2009 [bro10]. It is supported on both Linux and Windows on x86 and x86.64 processors.

C++CSP2 was developed because the original C++CSP lacked the ability to take full advantage of multicore processors; C++CSP2 supports true parallelism on multicore processors and multiprocessor systems.

C++CSP2 has timeouts and delays, but they are not hard real-time [c++10].

### 3.2.3 PyCSP

PyCSP is one of the newer CSP libraries; the first paper describing it was published in 2007, with the latest release being from 2009 [VBF09]. Its intended application is in scientific computing. Initially, PyCSP borrowed heavily from JCSP in terms of structure, but was simplified based on input from users (students). Thus, the current incarnation of PyCSP has only one channel type,
any2any, where both input and output channel ends support external choice. It also provides different types of CSP process implementations to overcome OS thread limits and the lack of parallelism caused by Python's Global Interpreter Lock (the GIL ensures exclusive access to Python objects).

PyCSP's platform compatibility is dependent on which process implementation method is chosen; threads are the most flexible, as they allow PyCSP to function effectively on all platforms supporting Python 2.6 or greater.

PyCSP has no Timed CSP support.

### 3.2.4 CSP.NET

CSP.NET was developed at the University of Copenhagen in 2006. It supports version 2.0 of Microsoft's .NET platform and has distributed support via .NET's Remoting and a workerpool Windows Service. Due to the lack of lightweight threads in the .NET API, CSP.NET uses true OS threads in a custom threadpool.

Unlike the other libraries, CSP.NET provides two different implementations of all CSP constructs: a local version and a distributed version. The architecture is otherwise similar to various iterations of JCSP et al.; it provides four channel types, prioritized and fair choice, and barriers.

CSP.NET implements a timeout event type for its choice construct.

The project appears to be inactive; its website (http://www.cspdotnet.com) no longer exists. CSP.NET was commercialized under the name of Jibu by spinoff company Axon7 [CKR08], but the company also appears to be defunct.
3.2.5 CHP

CHP (Communicating Haskell Processes) was developed in 2008 by Neil Brown (developer of C++CSP2 and contributor to JCSP). It was built to showcase the relative simplicity of writing concurrent code in a functional language rather than an imperative one.

CHP requires at least version 6.10 of the GHC Haskell compiler, and many common Haskell libraries [chp10].

CHP implements timeouts.

3.2.6 CTC++

CTC++ is a C++ library developed at the University of Twente by Bojan Orlic and Jan Broenink. It is targeted at the application area of real-time control systems [OB04]. To satisfy the requirements of real-time systems it uses a custom-built userspace process scheduler. It also has a similar feature set to the Kent libraries; it provides many CSP and CSP-like constructs, such as processes, a number of channel types, prioritized choice, prioritized parallelism, and barriers.

3.3 Code Synthesizers

These are systems that take CSP specifications as input and use them to produce code in a different language. Their aim is to produce code that is “correct by construction” – that retains the formal aspects of the (presumably verified) specifications used as input. The idea of code synthesis from a CSP specification is not a new one; Musha and Tokuda [MT83] described algorithms to convert CSP into coroutines nearly thirty years ago. Even so, code
synthesizers are not as common as formal libraries. Two systems of note were both created under the supervision of G. S. Stiles while he was at the University of Kent, while others were created by Jovanovic et al. at the University of Twente and Arrowsmith & McMillin at the University of Missouri-Rolla.

### 3.3.1 CC30

CC30 [AM94], created by Arrowsmith and McMillin in 1994, is a tool that translates a CSP program into C code. Each CSP process is converted to an OS process; channels between processes are implemented as sockets. All communicating processes must be on the same machine. Communication is underdeveloped; for instance, it is impossible to send constant values or functions across a channel. FDR2 and probe cannot be used on the CSP input, since the machine-readable CSP syntax used is not that of CSPm, and the CSP program must contain snippets of C code (such as a main() function and data types on processes) to be translated correctly.

### 3.3.2 CSP to JCSP

Raju and Stiles [RRS03] created tools to convert a CSP specification into Java code that makes use of the JCSP library. The CSPm syntax is used, enabling the use of ProBE and FDR2. The CSP features implemented are comments (line and block), channel declarations and I/O, integer data, external choice, synchronous parallelism, recursion, and semaphores. Plans were made to add support for floating-point data, but the project seems to have been abandoned; there have been no updates published since 2004.
3.3.3 CSPtoHC (CSP to Handel-C)

In 2004, Phillips and Stiles [PS04] created CSPtoHC, a tool to convert a CSPm specification into Handel-C code. The intended workflow (which is identical to CSP++’s own) is as follows: write a CSP specification, verify/correct with FDR2/ProBE, translate using CSPtoHC, optionally simulate using Handel-C simulator, edit, compile, program the FPGA.

The CSP features translated into Handel-C are comments (line and block), channel declarations and I/O, integer declarations, external choice, synchronous parallelism, and recursion. Notably absent from this version is the if-then-else CSP construct. Handel-C statements can be embedded in the CSP specification as comments.

Hand-editing of the Handel-C output is required if it is to be used on actual hardware; a clock source must be added, and channels that interface with external devices need to be mapped to valid FPGA port pins. If any optimization is required, it must also be done by hand; the translator does not perform any optimization of its Handel-C output.

Because the specification is written using CSPm syntax and any integrated Handel-C code is hidden in a CSP comment, ProBE and FDR2 can be used to verify the CSP program’s correctness.

3.3.4 gCSP

gCSP [BJ04] is a graphical tool for crafting GML (Graphical Modelling Language) [Hil02] representations of CSP systems. GML is similar in form and goals to UML, but is CSP-specific; it provides a standardized way to visually represent processes, channels, alts, I/O, etc. gCSP allows the creation of GML representations, and subsequent CSPm generation from them. The CSPm can
then be used for formal verification or for code generation by other tools. gCSP can also convert the GML directly into compilable code (Java, C++, oc-cam) by making use of CSP libraries (the CTC++ CSP library in particular) [BGL05].

gCSP does not support Timed CSP.

3.4 Tools Using CSP Internally

These are projects where the user needs no explicit knowledge of the work-ings of CSP; all formal models and specifications are hidden “under the hood”.

3.4.1 ASD

Analytical Software Design (ASD) is a suite of commercial tools created by Verum Software Technologies BV [ver10] that synthesizes code from math-ematically modelled requirements. The interface allows a user to specify a model’s states and transitions; the CSP process model is abstracted away. It generates C# or C++ code (support for Java, C, and VHDL is planned) from re-quirements and design specifications expressed in its own proprietary black-box format. These specifications can be refined and used to generate a CSP specification for formal verification via its own cloud-based verification suite. Furthermore, ASD can generate test cases from usage models derived from the requirements, run them, and analyze the results [BH10]. However, to build and execute ASD’s generated code, the ASD runtime must be present on the system.

ASD allows some integration of unverified user code via its Foreign Com-ponent system. Since ASD only uses CSP for verification, not system design and specification, Foreign Components are not as strongly mapped to CSP
events as CSP++ UCFs are. Additionally, they must also implement some ASD-specific interface methods.

ASD is compatible with Windows XP and Windows 7.

3.4.2 R2D2C

Requirements-to-Design-to-Code (R2D2C) [HRR05] is a project originating in NASA’s Software Engineering Laboratory that aimed to derive a CSP specification indirectly from a set of natural-language requirements. This is a multi-stage process: the requirements are converted into scenarios (human- and machine-readable descriptions of required system execution) [CG08], which would then be translated into CSP traces from which the (FDR2-compatible) CSP system specification is derived. Code can then be generated from the specification using existing (not necessarily R2D2C-specific) techniques.

R2D2C was never completed; Carter & Gardner’s work [CG08] was the first step in implementing the technique.
Chapter 4

CSP++

This chapter will familiarize the reader with the history and architecture of CSP++. This is important to put the present research in its proper context.

4.1 Overview

As previously mentioned, CSP++ is a software synthesis toolset that allows users to translate CSP specifications into executable C++ code, and to extend that generated code with user-coded C++ functions (UCFs). It consists of two components: the CSP translator (cspt), and the object-oriented application framework (OOAF).

The cspt translator accepts CSP specified in the CSPm dialect, for compatibility with the commercial CSP verification tools by Formal Systems Europe Limited. However, it deviates by also supporting Timed CSP, which is specified in a custom machine-readable notation [Sol08] due to the lack of timed operator support in CSPm. It outputs human-readable C++ code that uses OOAF classes to implement the input specification. These classes provide C++ implementations of the core CSP constructs — processes, events, and
channels — as well as CSP operator functionality.

The OOAF provides simple default behaviour for CSP events when they are not overridden with UCFs. UCFs may contain any code, but in order to preserve the correctness gained through verification of the source CSP specification, no attempt should be made to circumvent or subvert the formalism (for instance, by creating shared memory locations).

### 4.2 History

The first incarnation of CSP++ in its current form was created at the University of Victoria as part of W. Gardner’s dissertation [Gar00]. It has since undergone a number of significant changes by many contributors. The major points of interest in CSP++’s development history are outlined below.

#### 4.2.1 CSP Dialect Supported

In its early days, CSP++ supported specifications written in a homegrown machine-readable dialect. This dialect was written by Dr. M.H.M Cheng at the University of Victoria [Gar00] alongside a verification tool, csp12, which made use of it. Thus, the csp12 dialect was chosen for CSP++ in order to ensure compatibility with this tool.

In 2005 S. Doxsee [Dox05] reengineered CSP++ to replace csp12 support with support for the CSPm machine-readable CSP dialect instead. CSPm was more expressive and provided more features than the csp12 dialect, and was also the dialect of choice for the commercial verification tools created by Formal Systems (Europe) Limited [fse10]. Csp12’s remaining legacy lies largely in the CSP++ class nomenclature: the Agent and Action classes, which represent CSP process operations and CSP event operations respectively, take their
names from csp12 constructs. Some other reminders exist in the grammar rules that CSP++ translator uses to parse its input, but for the most part the source has been updated.

### 4.2.2 Platforms Supported

Initially, CSP++ (consisting of only the OOAF) was created on a Sun 4 Unix machine. It was intended to be built using AT&T's cfront C++ compiler and USL Standard Components Library, chosen for its support of multitasking and implementation of C++ templated data structures. However, the USL task library proved to be incompatible with the Sun 4 operating system, and so the required object files were cannibalized from an older version. Eventually, the first release of CSP++ was built using a combination of two different USL versions and the AT&T compiler on SunOS 4.

When the time came to distribute the project’s source, the need arose to port it to a more widespread platform. Red Hat Linux version 5.2 was chosen for its popularity on the x86 architecture. By this time, cfront and the USL library had been rendered obsolete by the GNU g++ compiler and the C++ Standard Template Library respectively, and the cspt translator had been written using these tools. The intention was to retain the tasking code from the previous version of CSP++ and simply recompile for the new platform, but that would have meant an assembly-level port of the old task code. Since inserting assembly code would have adversely affected the framework’s portability, the decision was made to do away with the old task model altogether and replace it with a threading model instead. The Red Hat implementation of POSIX threads, called LinuxThreads, was used to reimplement the core of the CSP++ task class while leaving the API much the same. Crucially, this resulted in a shift from AT&T's non-prioritized, non-preemptible coroutines
to preemptible threads, which necessitated the implementation of locks on some shared data structures. Because of this, there was never complete confidence in the correctness and completeness of this version of CSP++, leading to the search for a new, non-preemptible threading solution.

In 2004, the LinuxThreads implementation was swapped for the GNU portable threads package, Pth. Pth threads are userspace and therefore non-preemptible, and since Pth is an implementation of POSIX threads little of the framework had to be changed. Unfortunately, though use of Pth increased the portability of CSP++, it also increased program run times significantly [Dox05]. In addition, since Pth threads are scheduled in userspace, they cannot take advantage of modern multicore processors.

4.2.3 Micro CSP++

In 2006 a “micro” version of CSP++ (μcsp or ucsp), targeted at embedded and resource-limited systems, was created by J. Moore-Oliva [W. 09]. It differed from the “regular” CSP++ in that all usage of the memory-hogging C++ STL was removed and replaced with streamlined, non-templated, handwritten code. The C++ std::iostream classes were rewritten in a style similar to that of the standard C I/O implementation while retaining the C++ API, and packaged into a new ucsp namespace.

Regression tests, based on the CPPUnit unit testing framework and the Boost Test Library, were also introduced. Each test comprises a single, focused CSP specification and a record of its intended output. When run, the test harness compares the intended output to the actual output to verify that the test was successfully run. These tests have since been expanded upon with each new CSP++ modification. As of the time of writing there are 41 distinct tests.
The final addition was to create a build system using GNU autotools. This has since remained the method of CSP++ installation, as it allows anyone with a compatible Unix-variant platform (including the GNU g++ compiler and the GNU Pth threading library) to build it from source themselves. CSP++ has been open source since this version, and is distributed under the GNU GPL (for cspt) and LGPL (for the OOAF) licenses.

4.2.4 Timed CSP

CSP++ did not support Timed CSP until 2008 [Sol08], when the five currently-supported Timed CSP operators were implemented. This enabled CSP++ to be used for the synthesis of soft-realtime systems; previously, the toolkit could only be used on specifications that contained no timing information. This also introduced the use of C++ exceptions to the framework, as they were necessary to implement the interrupt operator; they were found to add little (about 1%) overhead to execution.

4.3 Design Flow

Figure 4.1 outlines the intended development process when using CSP++.

The process begins with the creation of a CSP specification written in the CSPm dialect. This is an ideal time to identify CSP events that should be attached to UCFs, and to begin UCF development. The CSP specification is processed by FDR2 and ProBE and any reported defects are corrected. This cycle of verification and correction continues until the specification is free of defects. At this point, cspt can be used to translate the specification from CSPm to C++ source code, which can then be compiled and linked with the CSP++ OOAF library.
If the system is in an early stage of development or does not require any UCFs, this may be an appropriate time to simulate and test it. All CSP events which are not linked to UCFs are bound to C++ stubs that provide some primitive default behaviour that can be useful for simulation and debugging: an atomic event prints its name, channel output prints the content of its data, and channel input prompts the user to input a value (currently only integer scalars are accepted). The executable can be run with the -t switch to print a trace during execution. If the system is further along in development or does make use of UCFs, any existing UCFs may be compiled into the system at this time.

An advantage of this design flow is that C++ developers can be productive even during the verification cycle, whereas with full formal specification it would be impractical to begin implementation before the entire system is verified.
4.4 Class Hierarchy

The following is an overview of the central classes in the CSP++ OOAF. For a graphical representation, Figure 4.2 depicts the CSP++ class structure as a UML class diagram.

- The Agent class wraps the legacy task class, and represents a schedulable thread of execution. An Agent executes functions of type AgentProc, each of which represents a CSP process as defined in the input specification. There does not necessarily exist a one-to-one mapping between Agents and CSP processes, since the translator generates additional Agents in some circumstances, such as breaking out subprocesses of complex parallel composition and generating code for sequential composition.

- The Action class is the base class of the Atomic and Channel classes, which represent CSP events and channels respectively.

- The Env class is the base class of the EnvSync, EnvSyncSet, EnvHide, EnvRename, and EnvInt classes. These represent an Agent’s environment objects, and use ActionRefs to manipulate Action instances with respect to other processes — for instance, by hiding or renaming them, or by setting them as process synchronization points.

For more detail on these classes, see Gardner’s dissertation [Gar00], Doxsee’s thesis [Dox05], and Solovyov’s thesis [Sol08]. The new classes introduced by the addition of process-specific storage, Storage and ICopyable, will be discussed in the next chapter.
Figure 4.2: CSP++ class diagram
Chapter 5

Process-Specific Storage

The main contribution of this research is to facilitate the full realization of selective formalism in CSP++, which has been strongly hampered until now. The idea of selective formalism [Gar00] starts from the use of CSP to model and formalize interprocess communication. This is CSP’s strength when creating abstractions of systems for verification purposes, or even when creating stripped-down systems for testing and simulation early in the development cycle, but a CSP specification does not contain enough information to be used as the sole source for the creation of a full-featured system. That is to say, the abstraction level of CSP alone is too high for most implementations, which is not surprising in view of the fact that it was never designed to be a programming language. Therefore, it is unsuitable for typical programming tasks like implementing common data structures and interfacing to third-party libraries.

The vision of selective formalism recognized the above by providing the UCF “escape hatch” from CSP for code that (a) is too awkward to express in CSP, and (b) does not need to be subjected to rigorous formal verification. In particular, it was always intended that it would be possible to replace by
UCFs, following verification and simulation, any of these specification elements:

- individual atomic or channel events
- groups of related events
- an entire process definition

Until now, the role for UCFs, practically speaking, was limited to the first case. This is because, for the latter two cases, a set of related UCFs would typically need to keep track of mutual state information and communicate amongst themselves, but the CSP++ framework provided no legitimate means to do so. This chapter describes how the problem is solved through making the framework provide so-called “process-specific storage”.

In Section 5.1, the need for, and rationale behind, process-specific storage is set out in more detail, with examples of what could not be done with existing CSP++. Section 5.2 discusses the theoretical issues raised by an inter-UCF communication mechanism, and how they were resolved. The implementation of process-specific storage is described in Section 5.3. Finally, Section 5.4 discusses the limitations of the present solution.

5.1 Rationale

As stated above, CSP was designed to model interprocess communication, not to provide all the facilities of a programming language. When translating CSP to source code, this usually means that we must work with either a severely abstracted or an overly-complex specification. These are both symptomatic of a common cause: a CSP specification cannot efficiently describe cooperation between events in a given process. This makes it difficult
for CSP to be used to specify and implement whole classes of applications that require such cooperation, such as complex user interfaces and artificial intelligence algorithms like neural networks.

The robot case study, discussed in Chapter 6, provides a good example. A PID (Proportional-Integral-Derivative) controller is a feedback loop, calculating some values and storing them for use in the loop’s next iteration. The following is a “functional model” of the PID system implemented:

```
1 PID = calc_error -> calc_integral -> calc_deriv -> calc_turn ->
   turn -> PID
2 SYS = PID
```

This is a simple, highly abstract representation of the system, and as such is suitable for use in formal verification and simulation. Ideally, we would be able to implement this system in CSP++ by simply replacing each atomic event with a UCF. However, this leads to a major problem: each event requires the use of values calculated in a previous event, and there is no way in CSP++ to pass these values directly along the process.

The way that CSP++ currently works, to create an implementation with UCFs, we would need to make two modifications:

- To get values into UCFs, we partition the functional model into stages, each of which is contained in its own process, to take advantage of the fact that UCFs can read process parameters.

- To get values out of UCFs, we convert the atomic events into channel events, which are capable of transmitting output.

The following “implementation model” illustrates how we accomplish this. Note how it uses the CSP control backbone to pass data among the UCFs; for example, the error value $e$. 

45
-- note: each calc_* event has an attached UCF that performs calculations and outputs the result

-- get an error value to be used later

PID_STAGE1(i,pe) = calc_error?e -> PID_STAGE2(i,pe,e)

-- using e: integral = integral + e
-- deriv = e - previous error

PID_STAGE2(i,pe,e) = calc_integral?ni -> calc_deriv?d ->
          PID_STAGE3(ni,e,d)

-- using e, i, d: turn = Kp * e + Ki * i + Kd * d
-- Kp, Ki, Kd are external constants available to the UCFs

PID_STAGE3(i,e,d) = calc_turn?t -> TURN(t,i,e)

-- make the turn, record this iteration’s i and e values, move to next iteration

TURN(t,i,e) = turn!t -> PID_STAGE1(i,e)

-- supply initial parameter values for system start

SYS = PID_STAGE1(0,0)

This approach has significant drawbacks.

Firstly, making the UCFs read process parameters is, strictly speaking, a “hack”. It was always intended that UCFs receive input from the CSP backbone via channel data, and the only reason that they can access process parameters directly is because those data members are declared “public” in the Agent class, so this is a loophole. It is a poor programming practice to
take advantage of it because the UCF’s access to the parameters is not visible at the CSP specification level; indeed, in PID_STAGE1 there is the illusion that the parameters i and pe are not used at all. UCF programmers should not be forced into such a recourse. That is to say, the CSP specification has its data — in process parameters and local variables — and the UCFs have theirs. When data crosses the boundary, then for the sake of transparency the transfer should be explicit, i.e., via channel I/O.

Secondly, the CSP specification is inflated and muddled by the input operations required to make the calculated values available as parameters and by the separation of the logical process into distinct CSP chunks purely to enable the parameter hack to work.

It is especially regrettable that creating the additional processes and variables — when they really are not needed in this application, because the control flow does not need to make decisions using their values — gratuitously increases the state space of the system. State space explosion is a challenge for verification tools such as FDR2 [For10], so it is always preferable to avoid expanding it whenever possible.

It’s worth noting that another potential approach to communicating data between UCFs may seem to exist; the FDR2 verification tool recognizes bidirectional channels, specified callit!a.b.c?x.y. This would send the contents of tuple a.b.c to callit, and return an output tuple in local variables x and y. CSP++ does not attempt this “mixed-mode” I/O; we currently treat channels as strictly input or output, as specified by Hoare [Hoa78], not both at the same time.

For this application, it would be much better if we could simply hook the atomic events of the original PID process to UCFs that communicated amongst themselves. Such communication would not violate the formalism,
because the same control flow will always be followed, as specified by the five abstract events.

At least there is a way to implement UCFs for the PID example using the current CSP++, awkward as it is. However, it only worked because the data in question could be put into integer form, which is presently the only type of data that can be passed as UCF arguments. Obviously, this will not always be the case, and so this approach is by no means a panacea. The disk request queue from the Disk Server Subsystem case study [Gar00] exemplifies this fact:

```plaintext
CELL = left?x.y -> shift -> right!x.y -> CELL

BUFF = ((CELL[[right<-comm]]) |{|comm|}) |{(CELL[[left<-comm]])}

DQueue = (DQ(0) |{|left, right, shift|}|) BUFF) \ |{|left, right, shift|}

DQ(2) = deq -> shift -> X(2)

DQ(i) = enq?x.y -> ( left!x.y -> shift-> DQ(i+1) )

[ ] deq -> ( if (i==0) then empty -> DQ(0)

else X(i) )

X(i) = right?y.z -> ( next!y.z -> DQ(i-1) )

[ ] shift -> X(i)
```

To fully model a queue (DQueue), we need two main processes that synchronize with each other: one to model the queue’s logic (DQ) and synchronize on external events (enq, deq, empty), and one to store the queue’s data.
(BUFF). CSP is well suited to model queue logic, but data storage is more naturally expressed in a programming language with built-in data structures. Moreover, data storage doesn’t need verification as it doesn’t affect the system’s control flow, so there’s no need to express it in CSP; the extraneous processes and events incur the same penalties outlined in the previous example.

Therefore, instead of modelling the queue’s data storage in the CSP specification, we can try to implement it directly with a C++ data structure in UCFs linked to the left, shift, and right events in DQ. However, this intended cooperation cannot occur unless the UCFs all have access to the same data structure. It is possible to have the UCFs communicate by writing to and reading from external files or global shared memory, but this behaviour and its consequences (inducing locks and critical sections on communication) are what CSP is intended to eliminate. Direct communication would be best, but cannot currently be accomplished.

What would make both this scenario and the PID example possible is for UCFs to have their own shared storage within CSP++ so that data created in one event can be made available to a subsequent one. To be maximally useful, the storage would be able to hold elements of any type — this would allow a programmer to work around CSP++’s present restriction to only integer data within the CSPm specification. To implement the PID example, we could attach UCFs to the events in the simple functional model that calculated the necessary values and used the shared storage to pass them along. To implement the queue, we could eliminate the BUFF and CELL processes and implement their storage functionality as a shared C++ queue. DQueue would be redefined as simply DQ(0) without the composition, and its internal events (left, shift, right) would be attached to UCFs that perform queue insertion.
and removal by having them call the C++ queue's appropriate member functions.

5.2 Theoretical Issues

There were some important points to consider when creating process-specific storage. Foremost among these were the issues of process identity and storage scope.

CSP process identity is not as straightforward as it may initially seem when looking at a written specification. Since a process is defined by its trace, a logical CSP “process” may be composed of multiple named processes (C++ AgentProcs) from the specification. Consider the following example.

```
P = foo -> Q
Q = bar -> SKIP
```

If we execute P successfully, the trace of the entire process up to termination will be <foo, bar>, which encompasses both P and Q. Now let us attach a UCF to foo and use it to store some data. This raises some questions: should Q be able to access that data? If so, should Q be able to modify that data? Will Q's changes be strictly local to Q, or will they propagate up and down the process chain?

This is the least complex scenario. Consider what happens when we introduce parallelism:

```
P = foo -> Q
Q = bar -> SKIP
SYS = P ||| P ||| P
```
We now have an additional question: should each invocation of \( P \) have its own distinct storage area, or should they all share a single one?

There is no simple, universal answer to these questions; various process structures may require different treatments. We consider various cases below.

The simplest case is one where the CSP specification is composed of only a single process, which terminates without chaining:

\[
P = \text{foo} \rightarrow \text{bar} \rightarrow \text{SKIP}
\]

In order for \( \text{bar} \) to access any data stored by \( \text{foo} \), they must both have access to some common storage location. The most sensible place to attach this storage is \( P \) itself; then any of \( P \)'s events can use it if they so desire. Thus, \( \text{foo} \) and \( \text{bar} \) can both store data in and retrieve data from \( P \)'s storage facility. In addition, \( \text{bar} \) can access and modify data that \( \text{foo} \) stored during its execution. There is no risk of breaking the formalism of the CSP specification here, as all communication is intraprocess only.

The more complex scenarios arise when a process chains or contains a sequential or parallel composition. These three different process structures must all be handled in different ways. In general, for process-specific storage to be useful, a process’ stored data needs to be made available to its component processes; this is in keeping with the concept of a logical CSP “process”.

A process chains by continuing its execution with another CSP statement. Because there is a continuous line of execution between the two statements, we want any data stored in the first process to migrate with it to the second when it chains. Consider the following:

\[
\begin{align*}
P &= \text{foo} \rightarrow Q \\
Q &= \text{bar} \rightarrow \text{SKIP}
\end{align*}
\]
Since a trace defines a process, and the trace of P consists of both foo (from P) and bar (from Q), we can allow Q to inherit any data stored in P when chained to. Because we can never return to P once we chain to Q, there is no need to preserve P’s stored data outside of Q. Therefore, it is safe for Q to inherit P’s storage directly, without the need for copying.

The CSP constructs that use chaining — i.e., where a process continues with its same or new identity — are:

- self-recursion
- mutual recursion
- choice
- timeouts

A self-recursive process chains onto itself, for example LIGHT = toggle -> LIGHT. This is a high-level functional model of a light that can be toggled off and on. In order for this specification to work as-is, the UCF attached to the toggle event needs to know and be able to set the state of the light. Therefore, if we set the light’s state in storage on one iteration of the process, it must be available to the subsequent iteration to be used. Then, the toggle UCF can look similar to this pseudocode:

```plaintext
toggle() {
    check state in storage
    if state is OFF
        turn light on
        set state ON in storage
    else
        turn light off
}
```
As with normal chaining, there is no need to preserve the state of any iteration other than the current one, since we can never “return” (as with a function call) to that line of execution once it has completed. Therefore, it is safe for its subsequent iteration to inherit its storage instance directly, without copying.

A self-recursive process may also be parameterized, as in this example from line 42 of our case study, listed in Appendix A.1 and discussed in Chapter 6:

```plaintext
CALIBRATE(i) = beep -500-> cal_s1 -> cal_s2 -> cal_s3 -5000->
   CALIBRATE(i-1)
```

The parameter has no effect on CALIBRATE’s execution or storage requirements. Since there is no special behaviour to account for, a parameterized self-recursive process may migrate its storage in the same manner as a non-parameterized self-recursive process.

This storage behaviour will allow solving DQueue as presented in section 5.1.

Mutually recursive processes chain to one another repeatedly. Here is another example from the case study:

```plaintext
NAV = report_l?l -> report_m?m -> report_r?r -> TURN(1,m,r)
 TURN(1,m,r) = calc_error -> calc_integral -> calc_deriv ->
   calc_turn -> turn_l -> turn_r -> NAV
```

NAV chains to TURN, and TURN then chains to NAV. The calc_integral UCF performs a calculation that uses the value it calculated in its previous iteration. In order for this value to be available when necessary, it needs to persist
through an invocation of NAV and get passed to TURN. Note the similarity in structure to self-recursion; this is, in fact, the same scenario. The name of the process we chain to is irrelevant; the stored data needs to reach the same point regardless of what it passes through. Because of this, we can treat mutual recursion the same way we treat self-recursion: the storage object of a process is inherited by the process it chains to.

Choice involves a process chaining to one of a number of possible processes, as demonstrated by LIGHTS below:

```
1 LIGHTS = ( toggle1 -> LIGHTS )
2
3   [] ( toggle2 -> LIGHTS )
4   [] ( toggle3 -> LIGHTS )
5
6   [] ( off -> ALL_OFF )
```

LIGHTS models three lights, each controlled by its own toggle switch as well as by a master off switch. As with the self-recursive example, the process must keep track of the lights’ states.

As can be seen, the process chained to may be the same as the original process, in which case the choice becomes a self-recursion, or it may be a different process, in which case the choice resolves to a normal chain. We know how to manage the storage of these constructs already — allow the chained process to directly inherit the storage of the process it originated in.

Timeouts are a variant of choice. Untimed timeouts, like choice, are resolved by an event, but timed timeouts are instead resolved by time elapsing. As this makes no difference to stored data, we can treat timeouts as we would treat choice.

More complex process structures involve process composition. These are:

- sequential composition
• interrupt

• parallel composition

Consider the sequential composition.

\[
\begin{align*}
P &= Q ; R \\
Q &= \text{foo} \rightarrow \text{SKIP} \\
R &= \text{bar} \rightarrow \text{SKIP}
\end{align*}
\]

A sequential composition is, in effect, two distinct processes executing consecutively within the sphere of a parent process. It is clear that both Q and R should have access to data stored in P’s storage facility, as P is the parent process of Q and R. The difficulty comes when trying to determine the relationship between Q and R themselves. Sequential composition differs from chaining (i.e. \( Q = \text{foo} \rightarrow R \)) in that Q is a sibling process to R, not its parent process. This implies that Q’s stored data should not propagate to R after Q terminates, since R’s execution does not originate in Q. However, according to the trace-based definition of a process stated above, Q and R are only continuations of the “logical process” P, and so R should have access to any data stored in Q.

These two views are obviously incompatible, so a compromise had to be made. We felt that the best way to reconcile these views would be to have Q and R behave as if they were each chained from P, one after another. This means that when Q begins, we give it its own storage area and copy P’s data by reference into it. Q can also store its own data in its storage. When Q terminates, any new data that Q stored is destroyed, but any changes made to P’s data remain. When R subsequently begins, it can inherit P’s data (including Q’s modifications) as it would from a normal chain, and R can operate on it at will, as well as store its own data. By making any data that originated
in $Q$ transient, we acknowledge the sibling relationship between $Q$ and $R$; by allowing $Q$'s modifications to $P$'s data to persist and so pass to $R$, we recognize that $Q$ and $R$ are children of $P$ and part of a greater whole.

An example of this structure can be seen in the $BOT\_RUN$ process on line 37 of our case study specification, where we needed storage objects created by its setup UCF to be available to the $GO$ process, which is the second process of a sequential composition with $CAL\_INIT$.

Interrupts are variants of sequential composition; the only difference is that the first process has no control over when it yields to the second. Since that decision does not affect storage, we can treat interrupts as we treat sequential composition.

The final process structure to consider is parallel composition. This includes both interleaved and synchronized parallelism; the type of parallelism involved does not affect the behaviour of process-specific storage, as the parent-child process relationships do not change between them. As an example, consider an interleaved parallel composition:

| P = Q || || R |
|--------------|-------------|
| Q = foo -> SKIP |
| R = bar -> SKIP |

Since $Q$ and $R$ execute simultaneously instead of sequentially, we avoid having to make the same compromise that we did with sequential composition. However, this parallel execution introduces its own complications. As in sequential composition, $P$ is the parent process of $Q$ and $R$. As such, $P$'s stored data must be made available to both processes, but if we simply copy references to $P$'s data as before, we introduce the possibility of both $Q$ and $R$ modifying the same piece of data at the same time. The solution to this problem is to copy the actual values of $P$'s data instead of references to it, but
this has the consequence of making all data modifications by Q and R strictly local; no changes are reflected in P’s storage. Unfortunately, this behaviour does contravene the “logical process” definition presented above, but since there is no other way to resolve this problem without imposing locks on data access (which CSP is intended to eliminate) or breaking the formalism of the CSP specification, we feel it is a reasonable price to pay.

Because storage migrates as processes transition, the copy-by-value necessary in parallel composition means that we must have a way of performing such copies on all stored data, even complex user-defined types. To be able to automatically clean stored data when it goes out of scope, we need a way to destroy the data that is accessible to the CSP++ framework.

The storage behaviour we have outlined is summarized in Table 5.1. In all cases, once storage goes out of scope, it is destroyed.

**Table 5.1: Proposed storage behaviour**

<table>
<thead>
<tr>
<th>process structure / operators</th>
<th>storage acquisition method</th>
</tr>
</thead>
<tbody>
<tr>
<td>self-recursion, mutual recursion, choice, timeouts</td>
<td>directly inherited from parent</td>
</tr>
<tr>
<td>sequential composition, interrupt</td>
<td>parent copied by reference to first process, directly inherited by second</td>
</tr>
<tr>
<td>interleaving, synchronized parallel</td>
<td>parent copied by value to all children</td>
</tr>
</tbody>
</table>

Another issue to consider was our choice of API. We would like a user to be able to store whatever data he wishes. If he stores a large number of items, he will need some simple way to differentiate them. It would be convenient for the user to be able to refer to his data as he would a C++ variable — by a name he defines. This way, a user can attach a known, meaningful label to a stored value. As with a C++ variable, the name would be tied to
the data for the life of the data, but the data’s value can change and still be attached to the same name. This mechanism will be familiar to a UCF programmer.

Since we want a user-defined name to be tied to a particular piece of stored data, it is natural to bind them together as a key-value pair. Storage for a process can consist of a set of these pairs, one pair per piece of data stored. A user can store and retrieve a particular piece of data using its associated name to insert or look up the pair in the set.

5.3 Implementation

Our implementation provides a process-specific storage (PSS) feature for UCFs that is similar to Pthreads’ and Pth’s thread-level storage (TLS) in both functionality and API. Its most important difference is its ability to automatically manage storage scope to match a UCF’s logical CSP process environment. By doing so, it allows UCFs to communicate with each other strictly within the context of a single CSP process — it keeps multi-process invocations of the same UCF from interfering with one another, and avoids breaking the formalism by facilitating interprocess communication outside of the controlling CSP specification.

This increases the practicality of selective formalism by boosting the power of UCFs. UCFs no longer need to force data transfers through the CSP backbone. Moreover, this was accomplished without requiring them to participate in choice, which was previously thought to be necessary.

Table 5.2 compares Pthreads’ and Pth’s thread-level storage to our process-specific storage implementation.

Pth’s thread-local storage could have been used as the basis for process-
Table 5.2: Comparison of process-specific storage and Pthreads'/Pth's TLS

<table>
<thead>
<tr>
<th></th>
<th>Pthreads</th>
<th>Pth</th>
<th>PSS</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>API</strong></td>
<td>thread_key_create, thread_key_delete, thread_getspecific, thread_setspecific</td>
<td>thread_key_create, thread_key_delete, thread_key_getdata, thread_key_setdata</td>
<td>Storage.get, Storage.set</td>
</tr>
<tr>
<td><strong>key format</strong></td>
<td>pthread_key_t (unsigned int)</td>
<td>thread_key_t (int)</td>
<td>const char*</td>
</tr>
<tr>
<td><strong>value format</strong></td>
<td>void*</td>
<td>void*</td>
<td>ICopyable*</td>
</tr>
<tr>
<td><strong>storage locations</strong></td>
<td>key: static struct; value: thread</td>
<td>key: static struct; value: thread</td>
<td>key, value: Storage object in Agent (thread)</td>
</tr>
<tr>
<td><strong>scope</strong></td>
<td>thread</td>
<td>thread</td>
<td>CSP process and descendants</td>
</tr>
</tbody>
</table>

Specific storage, as it provides a similarly structured storage facility and API. However, this would have a number of drawbacks:

- We would not be able to guarantee automatic cleanup of stored objects, as Pth stores values as void pointers and does not require users to supply destructors.

- We would still need to implement a way for any given value to be copied by value from thread to thread – again, because Pth stores values as void pointers and there is no way for a user to provide copy constructors, this is impossible to do with Pth's storage alone.

- It is possible that CSP++ will migrate to a new threading library in the future. Relying on Pth's storage is less flexible than having process-specific storage entirely self-contained.

The process-specific storage feature does not modify the CSP++ translator, but introduces a new class and an interface to the object-oriented application.
framework: Storage and ICopyable.

Storage is implemented as a C++ STL map of key-value pairs. The key is a user-defined C++ string that represents the data’s name or label, and the value is a pointer to an object that implements ICopyable. All objects stored must implement the ICopyable interface, whether they hold C++ primitives or complex user-created types. Some wrappers of C++ primitives have already been provided: IntCopyable, DoubleCopyable, and CharPtrCopyable hold ints, doubles, and char*s respectively.

Implementing ICopyable requires users to override the destructor and the copy() function, which should return a copied-by-value instance of this ICopyable. This allows automatic cleanup of storage on Agent destruction, and allows stored data to be properly migrated between processes when executing parallel compositions.

Every instance of the Agent class in the synthesized code has a Storage object attached. During Agent creation (which occurs in a STARTn macro call in the generated C++), its Storage object is also created; when the Agent is destroyed, so is its Storage object. This object is where all shared data that this Agent’s UCFs and constituent processes may access is stored.

As an Agent executes CSP processes, its Storage object is made available to them. The manner in which this occurs is determined by the execution context of the process.

If the process has been chained to (a CHAINn macro call in the generated C++), either from a normal chain or as part of a self-recursion, mutual recursion, choice, or timeout, the process can access and modify the Agent’s Storage object directly. Since, once we chain, we cannot return to the originating process, we do not need to restore the data’s previous state for future use.
Table 5.3: Agent branches values and corresponding process structures

<table>
<thead>
<tr>
<th>Value of branches</th>
<th>Process Structure</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>interrupt</td>
</tr>
<tr>
<td>1</td>
<td>sequential composition</td>
</tr>
<tr>
<td>2</td>
<td>parallel composition</td>
</tr>
</tbody>
</table>

This does not apply if the process is part of a sequential composition, interrupt, or parallel composition. Chaining does not spawn new Agents, but these constructs do. The child Agents cannot simply inherit the parent’s Storage because they may end up sharing it (in a parallel composition), leading to potential race conditions or invalid data, or exposing data outside of its proper scope (in a sequential composition or interrupt).

To solve this problem, we determine the Agent’s parental environment in its constructor, by examining its parent Agent’s branches variable. This variable records the number of subtasks an Agent will execute. Different values imply different process structures. Table 5.3 lists the possible structures and their corresponding branches values.

In a sequential composition ($P = Q ; R$), $P$ spawns $Q$ as a new Agent. When $Q$ terminates, $P$ chains to $R$. To implement the storage behaviour we described in Section 5.2, we first copy the contents of $P$’s Storage object by reference into $Q$’s Storage object. $Q$ can then modify that data at will, as well as add any new objects of its own. When $Q$ terminates, the objects that it stored are destroyed and their keys are removed from the Storage. The objects that were copied from $P$ remain. Since they were copied by reference from $P$, any changes made by $Q$ are reflected in $P$’s Storage. When $P$ chains to $R$, $R$ directly inherits $P$’s Storage as in a normal chain.

Interrupt ($P = Q \langle i \rightarrow R$, where $i$ is the interrupting event, or $P = Q \langle t \rightarrow R$, where $t$ is the time after which $R$ will begin) works similarly to
sequential composition. The only difference is that \( i \) and \( t \) have the ability to preempt the execution of \( Q \) (if \( i \) is immediately available for execution or \( t \) is zero), in which case \( P \) immediately chains to \( R \), with \( R \) inheriting \( P \)’s Storage object. Otherwise, \( P \) spawns \( Q \) as an Agent with a copy of \( P \)’s storage, and subsequently chains to \( R \), as in sequential composition.

In a parallel composition (\( P = Q || R \) or \( P = Q [ | \{ \text{foo} \}| ] R \)), \( P \) spawns both \( Q \) and \( R \) as new Agents. If we were to copy \( P \)’s Storage contents by reference into both \( Q \)’s and \( R \)’s Storage objects, any modification of that data would need to be placed in a critical section with locks around it — which is exactly the type of situation we want to avoid. Instead, we copy \( P \)’s stored data by value into \( Q \)’s and \( R \)’s Storages via ICopyable’s copy() method, which all implementations are required to override. \( Q \) and \( R \) operate independently on their local copies. When \( Q \) and \( R \) terminate, their respective stored data is destroyed. \( P \)’s data remains unchanged.

Storage cleanup is similarly dependent on execution context. If an Agent executes STOP, the CSP process can no longer continue, so all the ICopyables it contains are deleted. If an Agent executes SKIP (and so has its destructor called), there are two possibilities: if the Agent was spawned to be part of a sequential composition or interrupt, only the ICopyables associated with keys not present in the parent Agent’s Storage are deleted — we want to keep the parent’s Storage intact. Otherwise, the CSP process has terminated, so all ICopyables in the Agent’s Storage are deleted. In all cases, once the necessary ICopyables have been deleted, the Agent’s Storage object itself is also deleted.

Data is stored by calling Storage’s set method and passing it a user-defined name and a pointer to an ICopyable. The ICopyable’s internal value can be set by calling its setValue() method. Keys are unique; there can only be one instance of any given name in a Storage object. If a user tries to assign
a new value to a name that already exists in the storage, the old value will be overwritten.

Data is retrieved by calling get with the name of the desired stored data — we return a pointer to the associated ICopyable if the name exists in the Storage object, and NULL if it does not. To access the internal value of the ICopyable, call its getValue() method. A UCF developer should be aware of the true types of stored data, but can use C++’s dynamic_cast to test the validity of an attempted type conversion upon retrieval. This performs a check at run time to determine whether casting the object pointed to will yield a complete object of the requested type, and returns NULL if the answer is no. The example below demonstrates how to accomplish this.

```cpp
void get_ucaf(ActionType t, ActionRef* a, Var* v, Lit* l) {
    MyCopyable* mc = dynamic_cast<MyCopyable*>(THISAGENT->
        getStorage()--;get("mydata"));

    if (mc != NULL) {
        // do things with mc
    }
    else {
        // output of get() incompatible with MyCopyable*
    }
}
```

Unlike Pthreads and Pth, we do not provide a way to delete a key from the Storage object — instead, we provide automatic cleanup of stored data when it goes out of scope.

Appendix B provides a usage guide for UCF programmers who wish to utilize process-specific storage.
5.4 Limitations / Caveats

The `Storage` class reintroduces a dependency on the C++ Standard Template Library to CSP++, as it makes use of the STL map and string data structures. This is actually beneficial from the point of view of framework reliability, as the STL classes are almost certainly bug-free; however, when compiling a CSP++ program, statically linking in the standard C++ library adds approximately 45 KB to the executable’s size after stripping it of symbols. This has the potential to pose a problem in applications with heavy resource constraints. On the other hand, memory is cheaply available, even for small systems, so more real-world usage of CSP++ is required to determine if this is an actual problem. If it is, it can be easily remedied by reimplementing the `Storage` class with handwritten, STL-free map and string structures.

Because it is potentially problematic, users must enable process-specific storage explicitly by passing the `--enable-procstore` option to the configuration script when building CSP++. If the flag is not present, the process-specific storage feature will not be compiled.
Chapter 6

Case Study: LEGO PID-Controlled Line Follower

The existing case studies and test systems, such as the Disk Server Subsystem [Gar00], the Automatic Teller Machine [Dox05], and the VAC Automated Cleaner [Sol08], are good ways to demonstrate CSP++ and selective formalism, but are too-simple approximations of real-world systems. Since all their “environmental” interaction is simulated, they are entirely self-contained; there is no physical environment for the specified system to inhabit and interact with. This meant that the outward-facing features of CSP++ programs — the user-coded functions — were not being utilized to their full potential. Since the concepts of selective formalism and code extensibility are a large part of the CSP++ design philosophy, it was necessary to provide a suitable scenario in which user-coded functions could be properly exercised while simultaneously demonstrating the usefulness of the newly implemented per-process storage.

To this end, research was conducted into various robotic applications. Examples of considered applications include an extension of the robot vac-
uum cleaner case study [Sol08], a building-navigating robot using RFID tags as guides, and an arm-style robot that picked up and put down items. While interesting, most of these were discarded as infeasible due to lack of robotics or mechanical engineering experience. Eventually the decision was made to create a line-following robot using the LEGO Mindstorms NXT 2.0 kit, as it seemed to be a commonly used case study in the academic robotics world. Figure 6.1 depicts the robot we created.

![Figure 6.1: LEGO line-following robot](image)

The robot controller implements the PID (Proportional–Integral–Derivative) algorithm. After several unsatisfactory attempts to create our own customized line-following algorithm, in the end PID was chosen for both its suitability as a showcase for process-specific storage and for its widespread industrial use [Sel01]. PID has the additional benefit of not requiring choice or synchronization in its algorithm, which is useful as events with UCFs attached currently cannot participate in these constructs.
The controller sends instructions to a LEGO Mindstorms NXT 2.0 “Intelligent Brick”, which houses the kit’s processing components and runs version 1.28 of the LEGO NXT firmware. The brick is attached to the robot’s chassis, sensors, and motors with LEGO NXT parts. The robot controller runs on a host computer and uses the NXT++ library [nxt12] to interface with the LEGO NXT brick. Instructions are transmitted to the robot over USB. This is illustrated in Figure 6.2. It was necessary to modify the library so that it could make use of the new light sensors in the latest LEGO NXT robot kit.

![Figure 6.2: System diagram](image)

The LEGO robot makes use of three light sensors arrayed horizontally on the front of the robot’s chassis. Figure 6.3 illustrates the various possible positions of the sensor array over the line. The five arrangements in the left-hand column represent the ideal line-following scenarios, where the robot is more or less centered over the line. The four arrangements in the right-hand column represent the less-than-ideal situations, where the robot is not straddling the line, but is instead perpendicular to it or has lost sight of it altogether.

The PID implementation was tuned manually through trial and error. An attempt was made at tuning using the Ziegler-Nichols method [ZN42], but this did not yield useful results.

Section 6.1 describes the system’s design, Section 6.2 rationalizes our use of process-specific storage, and Section 6.3 details our verification of the specification using FDR2.
Figure 6.3: Sensor array positions for line detection cases
6.1 Design

This case study implements a line-following robot which, upon startup, will calibrate its light sensors and drive itself along a line on the ground, reading sensor data and adjusting its course accordingly until a switch on the robot is pressed. Besides the switch press, it only requires human intervention during the calibration phase, where the robot should be placed at various points along the line with its sensors appropriately positioned so that relevant readings can be made.

Whereas previous case studies have implemented a functional model to represent their target system and an environmental model to provide stimuli to that system, we provide a functional model only, and allow it to interact with the actual physical environment rather than a simulated one. Environmental models are useful tools that can facilitate rapid changes to a system during its design and testing phases by providing known, static input, but when the system is put to use in the real world, their usefulness rapidly declines. The CSPm specification listed below describes this functional model, and its associated UCFs can be found in Appendix A.1. Table 6.1 lists the UCFs present in the robot controller.

From this specification, it can be seen that there is a high degree of concurrency present in the controller: all the light sensors (SENSORS process) operate concurrently with the switch (SWITCH process) and the navigation systems (NAV process). This makes it a good candidate for specification in CSP and implementation with CSP++, since the purpose of selective formalism is to model concurrent systems.
Table 6.1: User-coded functions in robot case study

<table>
<thead>
<tr>
<th>CSP events</th>
<th>Functionality</th>
</tr>
</thead>
<tbody>
<tr>
<td>avg</td>
<td>estimate light level threshold for line based on calibration readings</td>
</tr>
<tr>
<td>beep</td>
<td>emit beep from NXT brick</td>
</tr>
<tr>
<td>cal_setup</td>
<td>initialize process-specific storage needed for sensor calibration</td>
</tr>
<tr>
<td>cal_s1, cal_s2, cal_s3</td>
<td>read individual light sensors for calibration</td>
</tr>
<tr>
<td>calc_error, calc_integral, calc_deriv, calc_turn</td>
<td>calculate and store the PID components</td>
</tr>
<tr>
<td>check_switch</td>
<td>check if switch has been pressed</td>
</tr>
<tr>
<td>close_bot</td>
<td>close connection to NXT brick</td>
</tr>
<tr>
<td>motors_off</td>
<td>turn both motors off</td>
</tr>
<tr>
<td>open_bot</td>
<td>open connection to NXT brick</td>
</tr>
<tr>
<td>read_l, read_m, read_r</td>
<td>read values from light sensors</td>
</tr>
<tr>
<td>setup</td>
<td>initialize process-specific storage needed for PID calculations</td>
</tr>
<tr>
<td>turn_l, turn_r</td>
<td>set power level of left and right motors</td>
</tr>
</tbody>
</table>

```csp
pragma cspt timeunit(ms)

channel open_bot : int
channel close_bot
channel setup, cal_setup
channel beep
channel cal_s1, cal_s2, cal_s3
channel avg
channel read_l, read_m, read_r : int
channel report_l, report_m, report_r : int
```
channel turn_l, turn_r
channel motors_off

channel calc_error, calc_integral, calc_deriv
channel calc_turn

channel check_switch : int
channel switch_pressed
channel delay

channel err

C cycles = 2

SYS = INIT

INIT = open_bot?opened -> if (opened == 1) then BOT_INT [ | { switch_pressed } | ] SWITCH
                      else err -> SKIP

BOT_INT = ( BOT_RUN \ switch_pressed -> SHUTDOWN )

BOT_RUN = setup -> CAL_INIT ; GO

CAL_INIT = cal_setup -> CALIBRATE(C cycles)

CALIBRATE(0) = avg -> SKIP

CALIBRATE(i) = beep -500-> cal_s1 -> cal_s2 -> cal_s3 -5000->
The controller has four main phases: initialization, calibration, line-finding, and shutdown. The initialization phase is the first to execute when the program is run. The INIT process (line 32) tries to open a connection to the NXT brick via a UCF attached to the open_bot event. If the connection attempt fails, the UCF returns a value of 0, which triggers the err event and subsequent termination of the process (and therefore the entire system). If the connection is successful, the UCF notifies the brick of the types of sensors attached to its ports (three light sensors and one touch sensor) and returns
a value of 1. This allows the commencement of the BOT_INT and SWITCH processes.

The SWITCH process (line 59) monitors the touch sensor with a UCF, periodically checking if it has been pressed. The SWITCH process runs in parallel with the controller's other operational processes. Also note that the BOT_RUN process (line 37) is interruptible by switch_pressed. This means that if the switch_pressed event is performed, it will bring the robot's normal operation to a halt and initiate the SHUTDOWN process (line 57) instead.

The setup event in BOT_RUN is tied to a UCF that creates the storage objects needed for PID calculations and robot steering:

- old_e, of type int, holds the error value from the previous PID iteration
- e, of type int, holds the error value from the current PID iteration
- i, of type double, holds the integral value from the current PID iteration
- d, of type int, holds the derivative value from the current PID iteration
- line, of type int, holds the threshold value for line detection
- turn, of type int, holds the calculated motor power adjustment for the current PID iteration

Once this is complete, the calibration phase begins. CAL_INIT (line 39) creates storage objects necessary during calibration in cal_setup, then begins CALIBRATE. The CALIBRATE process (lines 41 and 43) reads light sensor values for Ccycles iterations (where Ccycles is a CSPm integer constant), storing them cumulatively in the storage created in cal_setup, then averages them for each light sensor, yielding three separate results. The maximum of these three is chosen as the threshold value for line detection and is stored in the PID-specific storage slot named line. The CALIBRATE process terminates; the
storage that was created in cal_setup is destroyed as a result, since its scope has expired, and the controller enters the line-following phase with the start of the G0 process.

G0 (line 45) is composed of the SENSORS (line 47) and NAV (line 53) processes, synchronizing on communication over the report_l, report_m, and report_r channels.

SENSORS is itself a composition of three interleaved processes, one for each physical light sensor on the robot. SENSOR_L, SENSOR_M, and SENSOR_R (lines 49-51) each read the measured light values from their respective sensors and report them to the NAV process, repeatedly. The NAV process collects these reports and passes them along to the TURN process (line 55). TURN is where the PID algorithm is implemented:

- The calc_error event is attached to a UCF that calculates the error for this PID iteration based on the values of l, m, r, and the previously stored line. The error value is stored in the storage slot mapped to e.

- The calc_integral event is attached to a UCF that calculates the integral for this PID iteration from the error value in e and the accumulated integral value in i (zero for the first PID iteration). The newly calculated integral value is stored back into the i slot, overwriting the old value.

- The calc_deriv event is attached to a UCF that calculates the derivative for this PID iteration from the values of the current error (e) and the previous iteration’s error (old_e; zero for the first PID iteration).

- The calc_turn event is attached to a UCF that calculates the motor power adjustment for this PID iteration from the error, integral, and derivative values just calculated. The adjustment value is stored in the
slot named turn, and the value in old_e is overwritten by the current error in e.

- The turn_l and turn_r events are attached to UCFs that retrieve the stored power adjustment value from turn and adjust the motors’ power (and therefore the robot’s heading) appropriately.

Once both motors have been manipulated, the NAV process begins again, with the values just calculated still in storage and available to the next iteration. The robot controller continues executing in this manner until the touch sensor is pressed, at which point BOT_INT is interrupted and chains to the SHUTDOWN process. This turns off the motors and closes the connection to the NXT brick. When this process terminates, the storage objects and their contents are automatically destroyed by the CSP++ framework because their scope has ended.

It is important to recognize that this case study was carried out based on the existing simple framework-to-UCF interface. This proved possible because PSS enabled UCFs executing in the same process to cooperate at the data level. This meant that it was not necessary for UCFs to force their data into integer format and pass it awkwardly through the CSP backbone, nor did the backbone have to exercise complex control over the UCFs. In this way, the addition of PSS increased the power of CSP++ without making the UCF interface more elaborate.

Since GNU Pth uses thread aging to avoid starvation [pth12], CSP processes that may at first glance appear to be monopolizing the CPU, such as the SENSOR_L, SENSOR_M, and SENSOR_R processes, are actually not doing so.
6.2 Use of Process-Specific Storage

There are three main reasons that the PID algorithm implemented in the robot controller is a good choice to demonstrate the use of process-specific storage. Firstly, the PID algorithm is iterative, and each iteration is mainly composed of sequential calculations where a given operation requires the result of a previous one. It was not previously possible to pass these values between user-coded functions in CSP++. Secondly, the PID algorithm is a feedback loop: calculations in a given iteration also require the use of values calculated in the previous iteration. As mentioned in the introduction of this thesis, this is possible through some unorthodox manipulation of processes, but no solution that retained process identity existed until now. Thirdly, those values are only used for internal calculations and do not affect the state of the system’s control backbone. This means that specifying the calculations in CSP would complicate the specification and inflate its state space, but not provide any benefit.

The utility of selective formalism can be demonstrated by comparing the lengths of the CSPm specification and the set of C++ UCFs. As can be seen in above and in Appendix A, the specification is 59 lines long, while the UCFs together are 206 lines long. This illustrates the real power of selective formalism – with the majority of the non-critical, data-centric code residing in UCFs, the formally modelled CSP backbone remains simple and straightforward.
6.3 Verification

The robot controller’s specification makes use of two Timed CSP operators: untimed interrupt and timed prefix. Ideally, to verify a specification that incorporates Timed CSP, we would use a verification tool that can accommodate its timed operators, such as the aforementioned HORAE. Unfortunately, HORAE is unavailable for use at this time.

Instead, we use FDR2. FDR2 can handle untimed interrupt, and we can replace the timed prefixes in our specification with standard CSP prefixes. This is valid because the timed prefixes in our specification do not affect opportunities for synchronization between processes and so will not affect the system’s transitions between states – for other specifications, e.g., ones also using timed interrupt or timed timeout, this may not be the case.

6.3.1 Deadlock, Livelock and Determinism Checks

First we use FDR2 to ensure that our system will not unexpectedly enter a deadlocked or livelocked state. In order to make these checks feasible, we gave channels that were of type Int smaller, finite ranges to shrink the system’s state space. This does not invalidate the checks because the values transmitted over those channels do not affect the system’s execution path.

One other modification is necessary in order to perform the deadlock check. When FDR2 encounters SKIP, it treats the system as having entered a deadlock because it is not able to progress any further. However, this is actually the desired system behaviour — when we execute SKIP, the system terminates and should not perform any more events. FDR2 cannot evaluate the system for true deadlocks until these termination “deadlocks” are out of the way. To eliminate them, we replace both instances of SKIP with a dummy
process whose only action is to infinitely recurse. This does not affect the system’s execution path in terms of true deadlocks; the only change is that the system gets stuck in an infinite recursion instead of terminating normally.

The livelock check does not require the changes we instituted for the deadlock check, or indeed any other changes beyond the reduction of Int channels’ ranges.

With these modifications, which are shown in the CSP snippets below, FDR2 proves that our robot controller is both deadlock and livelock free. The FDR2 result windows for both tests are below the specification.


dummy

```
-- reduced range of channel types for read_*, report_ *
channel read_l, read_m, read_r : {0..50}
channel report_l, report_m, report_r : {0..50}

-- new recursive DUMMY process (used for deadlock check)
DUMMY = DUMMY

-- original INIT process def’n (used for livelock check)
INIT = open_bot?opened -> if (opened == 1) then BOT_INT [ ] [ 
  switch_pressed | ] ] SWITCH 
    else err -> SKIP

-- modified INIT process def’n (used for deadlock check)
INIT = open_bot?opened -> if (opened == 1) then BOT_INT [ ] [ 
  switch_pressed | ] ] SWITCH 
    else err -> DUMMY

-- original SWITCH process def’n (used for livelock check)
SWITCH = check_switch?pressed -> if (pressed == 1) then 
  switch_pressed -> SKIP
```
else delay -> SWITCH

-- modified SWITCH process def'n (used for deadlock check)

SWITCH = check_switch?pressed -> if (pressed == 1) then
          switch_pressed -> DUMMY
          else delay -> SWITCH

Figure 6.4: FDR2 deadlock test results

Figure 6.5: FDR2 livelock test results

We also used FDR2 to confirm that our system is deterministic; that is, that its execution paths are always predictable and unchanging. This is a useful property to have when generating code from a specification because it implies that the code should always produce the same result.
6.3.2 Trace Refinement

Next, we capture some traces generated during the system’s execution and use FDR2 to confirm that they refine our main system process, SYS. This aims to show that the observed sequence of events executed by the system is, in fact, allowed by the CSP specification, and so that our translation from CSP to executable code is a faithful one. Of course, since we are not capturing all possible traces, even if the traces we do capture refine SYS this is not definite proof that our implementation is correct. However, it will provide a greater measure of confidence in our translation. We made use of Solovyov’s Python script [Sol08] to convert our CSP++ traces to FDR2-compatible CSPm.

Our first trace, produced automatically by simply executing the robot program with the “-t” command line option, lists the sequence of events that occurs when the robot controller cannot open a connection to the robot itself. Each event, including the specific channel data, is displayed between brackets [ ] and is preceded by the name of the process performing the event. If multiple processes synchronize on an event, the framework’s convention is to identify the one that arrived last at the synchronization.
|=INIT [open_bot.0]
|=INIT [err]

The Python script translates this to CSPm and generates an FDR2 assertion to test for trace refinement:

```plaintext
--accumulated trace
TRACE = open_bot.0 -> err -> STOP

--FDR assertion
assert SYS [T= TRACE]
```

Through running this assertion, FDR2 confirms that this trace is a refinement of SYS, as can be seen in Figure 6.7.

![Figure 6.7: FDR2 trace 1 refinement check results](image)

Our second trace lists events that occur during sensor calibration. We could not use the entire trace we captured, as its translated CSPm form was too long for FDR2 to parse. FDR2's maximum line length is not documented, and breaking a process over multiple lines (without changing its structure by sequentially composing subprocesses) is, to the best of our knowledge, not possible. Sensor calibration takes at least ten seconds, most of which is
spent with the CALIBRATE process waiting and the SWITCH process repeatedly executing check_switch.0 -> delay between calibration events. To minimize the trace, we removed most of the repetitive switch-checking events, as they do not affect the system’s execution. The resulting process is shown below.

```plaintext
1 | =INIT [open_bot.1]
2 | =BOT_RUN [setup]
3 | =CAL_INIT [cal_setup]
4 | =CALIBRATE( 2 ) [beep]
5 | =SWITCH [check_switch.0]
6 | =SWITCH [delay]
7 | =CALIBRATE( 2 ) [cal_s1]
8 | =CALIBRATE( 2 ) [cal_s2]
9 | =CALIBRATE( 2 ) [cal_s3]
10 | =SWITCH [check_switch.0]
11 | =SWITCH [delay]
12 | =CALIBRATE( 1 ) [beep]
13 | =SWITCH [check_switch.0]
14 | =SWITCH [delay]
15 | =CALIBRATE( 1 ) [cal_s1]
16 | =CALIBRATE( 1 ) [cal_s2]
17 | =CALIBRATE( 1 ) [cal_s3]
18 | =SWITCH [check_switch.0]
19 | =SWITCH [delay]
20 | =CALIBRATE( 0 ) [avg]
```

This is translated to:

```plaintext
--accumulated trace
```
FDR2 confirms that this trace is also a refinement of SYS.

Our next trace lists events that occur when the robot makes a lap around an oval track. As above, we could not use the entire trace, as its CSPm form was over 412,000 characters long. We again removed consecutive switch checks during the calibration phase, as well as a large portion of the events in the line-following phase.
\begin{verbatim}
| = SWITCH [check_switch.0]
| = SWITCH [delay]
| = CALIBRATE( 2 ) [cal_s1]
| = CALIBRATE( 2 ) [cal_s2]
| = CALIBRATE( 2 ) [cal_s3]
| = SWITCH [check_switch.0]
| = SWITCH [delay]
| = CALIBRATE( 1 ) [beep]
| = SWITCH [check_switch.0]
| = SWITCH [delay]
| = CALIBRATE( 1 ) [cal_s1]
| = CALIBRATE( 1 ) [cal_s2]
| = CALIBRATE( 1 ) [cal_s3]
| = SWITCH [check_switch.0]
| = SWITCH [delay]
| = CALIBRATE( 0 ) [avg]
| = SENSOR_L [read_l.441]
| = SENSOR_M [read_m.200]
| = SENSOR_R [read_r.436]
| = SENSOR_L [report_l.441]
| = SENSOR_L [read_l.441]
| = NAV [report_m.200]
| = NAV [report_r.436]
| = TURN( 441, 200, 436 ) [calc_error]
| = TURN( 441, 200, 436 ) [calc_integral]
| = TURN( 441, 200, 436 ) [calc_deriv]
| = TURN( 441, 200, 436 ) [calc_turn]
| = TURN( 441, 200, 436 ) [turn_l]
| = TURN( 441, 200, 436 ) [turn_r]
\end{verbatim}
| =NAV [report_l.441] |
| =SENSOR_M [read_m.200] |
| =SENSOR_M [report_m.200] |
| =SENSOR_M [read_m.200] |
| =SWITCH [check_switch.0] |
| =SWITCH [delay] |
| =SENSOR_R [read_r.439] |
| =SENSOR_L [read_l.439] |
| =NAV [report_r.439] |
| =TURN( 441, 200, 439 ) [calc_error] |
| =TURN( 441, 200, 439 ) [calc_integral] |
| =TURN( 441, 200, 439 ) [calc_deriv] |
| =TURN( 441, 200, 439 ) [calc_turn] |
| =TURN( 441, 200, 439 ) [turn_l] |
| =TURN( 441, 200, 439 ) [turn_r] |
| =NAV [report_l.439] |
| =NAV [report_m.200] |
| =SENSOR_R [read_r.443] |
| =SENSOR_R [report_r.443] |
| =SENSOR_R [read_r.439] |
| =SWITCH [check_switch.0] |
| =SWITCH [delay] |
| =SENSOR_L [read_l.443] |
| =SENSOR_M [read_m.201] |
| =TURN( 439, 200, 443 ) [calc_error] |
| =TURN( 439, 200, 443 ) [calc_integral] |
| =TURN( 439, 200, 443 ) [calc_deriv] |
| =TURN( 439, 200, 443 ) [calc_turn] |
| =TURN( 439, 200, 443 ) [turn_l] |
| =TURN( 439, 200, 443 ) [turn_r]  |
| =NAV [report_l.443]             |
| =NAV [report_m.201]             |
| =NAV [report_r.439]             |
| =TURN( 443, 201, 439 ) [calc_error] |
| =TURN( 443, 201, 439 ) [calc_integral] |
| =TURN( 443, 201, 439 ) [calc_deriv] |
| =TURN( 443, 201, 439 ) [calc_turn] |
| =TURN( 443, 201, 439 ) [turn_l]  |
| =TURN( 443, 201, 439 ) [turn_r]  |
| =SENSOR_L [read_l.443]         |
| =SENSOR_L [report_l.443]       |
| =SENSOR_L [read_l.444]         |
| =SWITCH [check_switch.0]       |
| =SWITCH [delay]                |
| =SENSOR_M [read_m.201]         |
| =SENSOR_R [read_r.441]         |
| =NAV [report_m.201]            |
| =NAV [report_r.441]            |
| =TURN( 443, 201, 441 ) [calc_error] |
| =TURN( 443, 201, 441 ) [calc_integral] |
| =TURN( 443, 201, 441 ) [calc_deriv] |
| =TURN( 443, 201, 441 ) [calc_turn] |
| =TURN( 443, 201, 441 ) [turn_l]  |
| =TURN( 443, 201, 441 ) [turn_r]  |
| =NAV [report_l.444]            |
| =SENSOR_M [read_m.198]         |
| =SENSOR_M [report_m.198]       |
| =SENSOR_M [read_m.198]         |
|=SWITCH [check_switch.0]
|=SWITCH [delay]
|=SENSOR_R [read_r.441]
|=SENSOR_L [read_l.441]
|=NAV [report_r.441]
|=TURN( 444, 198, 441 ) [calc_error]
|=TURN( 444, 198, 441 ) [calc_integral]
|=TURN( 444, 198, 441 ) [calc_deriv]
|=TURN( 444, 198, 441 ) [calc_turn]
|=TURN( 444, 198, 441 ) [turn_l]
|=TURN( 444, 198, 441 ) [turn_r]
|=NAV [report_l.441]
|=NAV [report_m.198]
|=SENSOR_R [read_r.440]
|=SENSOR_R [report_r.440]
|=SENSOR_R [read_r.440]
|=SWITCH [check_switch.1]
|=SWITCH [switch_pressed]
|=SENSOR_L [read_l.490]
|=SENSOR_M [read_m.172]
|=TURN( 490, 182, 386 ) [calc_error]
|=TURN( 490, 182, 386 ) [calc_integral]
|=TURN( 490, 182, 386 ) [calc_deriv]
|=TURN( 490, 182, 386 ) [calc_turn]
|=TURN( 490, 182, 386 ) [turn_l]
|=TURN( 490, 182, 386 ) [turn_r]
|=NAV [report_l.490]
|=NAV [report_m.172]
|=NAV [report_r.381]
This is translated to:

```plaintext
TRACE = open_bot.1 -> setup -> cal_setup -> beep -> check_switch.
  .0 -> delay -> cal_s1 -> cal_s2 -> cal_s3 -> check_switch.0 ->
  delay -> beep -> check_switch.0 -> delay -> cal_s1 -> cal_s2
  -> cal_s3 -> check_switch.0 -> delay -> avg -> read_l.1 ->
  read_m.0 -> read_r.6 -> report_l.1 -> read_l.1 -> report_m.0
  -> report_r.6 -> calc_error -> calc_integral -> calc_deriv ->
  calc_turn -> turn_l -> turn_r -> report_l.1 -> read_m.0 ->
  report_m.0 -> read_m.0 -> check_switch.0 -> delay -> read_r.9
  -> read_l.9 -> report_r.9 -> calc_error -> calc_integral ->
  calc_deriv -> calc_turn -> turn_l -> turn_r -> report_l.9 ->
  report_m.0 -> read_r.3 -> report_r.3 -> read_r.9 ->
  check_switch.0 -> delay -> read_l.1 -> read_m.1 -> calc_error
  -> calc_integral -> calc_deriv -> calc_turn -> turn_l ->
  turn_r -> report_l.13 -> report_m.1 -> report_r.9 -> calc_error
  -> calc_integral -> calc_deriv -> calc_turn -> turn_l ->
  turn_r -> read_l.13 -> report_l.13 -> read_l.4 -> check_switch.0
  -> delay -> read_m.1 -> read_r.1 -> report_m.1 -> report_r.
  1 -> calc_error -> calc_integral -> calc_deriv -> calc_turn ->
  turn_l -> turn_r -> report_l.4 -> read_m.8 -> report_m.8 ->
```
Unlike the previous traces, this one is not a refinement of SYS. The reason is that the BOT_INT process continues to execute events for a time after the interrupting event switch_pressed occurs instead of immediately terminating and continuing with SHUTDOWN, as can be seen in this subset of the example trace that FDR provided:

```plaintext
open_bot.1
setup
...
check_switch.1
switch_pressed
read_l.1
read_m.2
read_r.0
check_switch.1
assert SYS [T= TRACE

This is most likely due to Pth’s scheduling algorithm. The left-hand side of the interrupt is terminated and the thread executing its parent CSP process, which was previously sleeping, is woken up and continues as the right-
hand side. The interrupt operator’s functionality is based on waiting for and signalling Pth’s condition variables. To execute the right-hand side of an interrupt, we rely on the Pth scheduler running the thread whose condition variable has been signalled. However, there is no guarantee that the scheduler will run the signalled thread over any other waiting thread [Sol08]. We try to influence the scheduler by maximizing the desired thread’s priority, but otherwise thread scheduling is beyond our control.

Through investigation we have seen that the signal to wake the thread does occur promptly after the switch has been pressed. However, the scheduler chooses to wake another waiting thread instead of the desired thread, leading to the behaviour we have observed. Once the scheduler allows the desired thread to run, the SHUTDOWN process occurs immediately.

### 6.3.3 Summary

The above verification steps prove that the CSP specification is deadlock free, livelock free, and deterministic. Trace refinement shows that the specified behaviour of the implementation was not damaged by its execution of many unverified UCFs. The only issue we encountered was the delayed execution
of the interrupt operator, which is a Pth thread scheduling issue and not the fault of the UCFs. Future work will be needed to determine whether the framework can exert better control over the Pth scheduler in the case of interrupts.
Chapter 7

Performance Metrics

Determining the effects that the addition of process-specific storage has on the performance of programs created using the CSP++ framework is essential. The following tests compare program run times and memory usage between CSP++ versions. They also investigate the effects of static library linking on program run times and memory usage, since the new storage features introduce some C++ STL classes into the framework which have the potential of increasing the program’s memory size.

All tests were performed on a Intel(R) Core(TM) i5-2500K 3.30GHz processor with 3GB of memory, running Gentoo Linux with a vanilla version 3.2.1 kernel. The compiler and threading library used were g++ 4.5.3 and GNU Pth 2.0.7. Compiler optimization was set at level 2 (-O2).

Test executables were run with the CSP++ “quick stop” flag (-q) to suppress the usual text dump that occurs when a process executes STOP. Nothing else was written to the standard output stream to avoid unnecessary run time inflation.

When comparing CSP++ versions to determine the effect of PSS, the official branch of the previous version (5.0) was not used for testing. The reason
for this is that bug fixes and refinements were implemented after the branch was made and were not merged back down into the older version. In order to take advantage of these changes, the storage features were enabled via a configure flag in one current CSP++ installation, and disabled in another. These two installations were then used for the comparisons.

The test system used, the Disk Server Subsystem case study written by W. Gardner [Gar00], is the de facto benchmark system for CSP++ modifications. Previous contributors have all used variations of the DSS system to measure the effects of their changes. The existing demo scenario in the DSS case study was replaced with four variants, adapted from S. Doxsee [Dox05] and Y. Solovyov [Sol08] with slight modifications and specified below. No UCFs were involved in this benchmark; it is just the CSP++ framework executing the specification by itself.

Variant 1 simulates 2 clients performing 20000 disk accesses in 20000 processes.

```csharp
C(1) = ds!1.100 -> ack.1->SKIP
C(2) = ds!2.150 -> ack.2->SKIP
TEST(i) = if (i>0) then ((C(1) || C(2)); TEST(i-1)) else STOP
SYS = (DSS [{ds,ack}] TEST(10000)) \ {{dint,dio}}
```

Variant 2 simulates 2 clients performing 20000 disk accesses in 2 processes.

```csharp
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(10000)) \ {{dint,dio}}
```
Variant 3 simulates 2 clients performing 10000 disk accesses in 10000 processes.

\[
\begin{align*}
C(1) &= \text{ds!1.100} \rightarrow \text{ack.1} \rightarrow \text{SKIP} \\
C(2) &= \text{ds!2.150} \rightarrow \text{ack.2} \rightarrow \text{SKIP} \\
\text{TEST}(i) &= \text{if } (i>0) \text{ then } ((C(1) \mid \mid C(2)) ; \text{TEST}(i-1)) \text{ else STOP} \\
\text{SYS} &= (\text{DSS [][ds,ack]}\mid\mid \text{TEST}(5000)) \setminus \{\text{dint, dio}\}
\end{align*}
\]

Variant 4 simulates 4 clients performing 20000 disk accesses in 4 processes.

\[
\begin{align*}
C(1,n) &= \text{if } n > 0 \text{ then } \text{ds!1.100} \rightarrow \text{ack.1} \rightarrow C(1,n-1) \\
&\quad \text{else SKIP} \\
C(2,n) &= \text{if } n > 0 \text{ then } \text{ds!2.150} \rightarrow \text{ack.2} \rightarrow C(2,n-1) \\
&\quad \text{else SKIP} \\
C(3,n) &= \text{if } n > 0 \text{ then } \text{ds!3.200} \rightarrow \text{ack.3} \rightarrow C(3,n-1) \\
&\quad \text{else SKIP} \\
C(4,n) &= \text{if } n > 0 \text{ then } \text{ds!4.250} \rightarrow \text{ack.4} \rightarrow C(4,n-1) \\
&\quad \text{else SKIP} \\
\text{TEST}(i) &= ((((C(1,i) \mid \mid C(2,i)) \mid \mid C(3,i)) \mid \mid C(4,i)) ; \text{STOP} \\
\text{SYS} &= (\text{DSS [][ds,ack]}\mid\mid \text{TEST}(5000)) \setminus \{\text{dint, dio}\}
\end{align*}
\]

The majority of the experiments were performed using Variant 3 as it had the shortest run time and was therefore the most convenient to execute repeatedly.

### 7.1 Effect of Process-Specific Storage

With these tests we aimed to determine whether the overhead introduced by the addition of process-specific storage to CSP++ was within acceptable levels. For run time, we considered an increase of up to ten percent to be
acceptable; this number allows for the fluctuations in run time increases between test variants due to their different process configurations. For memory usage, we considered an increase of up to 100KB acceptable, taking into account the fact that we introduced a new class with a C++ Standard Template Library dependency.

We hypothesized that our tests would show that the overhead incurred was indeed within the acceptable range.

### 7.1.1 On Run Time

To determine the effect of process-specific storage on the run times of CSP++ programs, each test variant above was compiled with both the storage-free and storage-implemented versions of CSP++ and run 30 times. The GNU time utility (not the shell built-in) was used to measure the execution time of each run. The data was determined to be normally distributed via visual inspection of histograms as well as the three-sigma test. The measured run times were then averaged and the results recorded in Table 7.1.

With storage, the highest standard deviation of a batch of 30 runs is 0.48 seconds. Without storage, the highest standard deviation of a batch of 30 runs is 0.41 seconds.

Table 7.1: Average CSP++ program run times (s), with and without storage

<table>
<thead>
<tr>
<th>Variant</th>
<th>Variant 1</th>
<th>Variant 2</th>
<th>Variant 3</th>
<th>Variant 4</th>
</tr>
</thead>
<tbody>
<tr>
<td>with storage</td>
<td>13.46</td>
<td>16.94</td>
<td>6.68</td>
<td>21.81</td>
</tr>
<tr>
<td>without storage</td>
<td>13.20</td>
<td>16.19</td>
<td>6.49</td>
<td>21.02</td>
</tr>
<tr>
<td>difference</td>
<td>0.26</td>
<td>0.75</td>
<td>0.19</td>
<td>0.79</td>
</tr>
</tbody>
</table>

It can be seen that the impact of process-specific storage on CSP++ program run time is noticeable, but not significantly large, and within the ac-
ceptable range. Run time increases range from about one-fifth to four-fifths of a second. Variant 2’s run time had the largest percentage increase, at 4.6%. The smallest percentage increase was 1.9% on Variant 1.

The DSS case study was not modified to make use of process-specific storage in UCFs, so the increase in run time can be attributed to the creation of the empty storage structures when Agents spawn and chain in the synthesized C++ code.

### 7.1.2 On Memory Usage

To determine the effect of process-specific storage on the memory usage of CSP++ programs, test variant 3 was compiled with both the storage-free and storage-implemented versions of CSP++, using static linking on libstdc++ and stripping the executables of symbols each time. The file sizes of the resulting executables are used as rough estimates of their memory usage, and are recorded in Table 7.2.

**Table 7.2:** Estimated CSP++ program memory usage (KB), with and without storage

<table>
<thead>
<tr>
<th></th>
<th>Variant 3</th>
</tr>
</thead>
<tbody>
<tr>
<td>with storage</td>
<td>167</td>
</tr>
<tr>
<td>without storage</td>
<td>122</td>
</tr>
<tr>
<td>difference</td>
<td>45</td>
</tr>
</tbody>
</table>

Since the process-specific storage feature is the only part of the CSP++ OOAF that makes use of the C++ STL, the difference in file size can be directly attributed to the presence of the standard C++ libraries used in its implementation. It can be seen that using C++ STL data structures adds a fixed overhead of approximately 45 KB to a CSP++ program’s memory usage. This is not unreasonable; for small test cases like the DSS variants this overhead
amounts to about one-quarter of the total file size, but for a more complex program this proportion will shrink. If the program uses UCFs that also make use of the C++ STL, the overhead introduced by the process-specific storage will be irrelevant.

This overhead is well within the acceptable range. However, in an environment with limited resources such as an embedded system, where almost any overhead is unacceptable, the overhead that PSS incurs may be too large. If this is the case, storage data structures can be rewritten to avoid reliance on STL classes; the underlying algorithm can remain the same.

### 7.1.3 As an Alternative to “Ad-hoc” Storage

To get a sense of whether PSS is a light-weight or heavy-weight mechanism, we extracted the PID calculation process and UCFs from the case study in Chapter 6 and put them in a loop for one million iterations. We then replaced the PSS operations in the UCFs with operations on global variables, such as a UCF programmer may have resorted to using before the PSS feature was introduced.

The CSP for this is shown below.

```
channel setup
channel calc_error, calc_integral, calc_deriv, calc_turn

SYS = setup -> PROC(1000000)

PROC(0) = SKIP

PROC(n) = calc_error -> calc_integral -> calc_deriv -> calc_turn
          -> PROC(n-1)
```
The only difference between the UCFs used in the first test and the UCFs used in the case study is that here, the error variable is initialized with a value of 1 instead of being generated from sensor data; therefore, as they can be seen in Appendix A.1, the first test's UCFs are omitted here. The second test's UCFs, modifying global variables, are shown below.

```c
int old_e, e, d, turn;
double i;

void setup_proc(ActionType t, ActionRef* a, Var* v, Lit* l) {
    old_e = 0;
    e = 0;
    i = 0;
    d = 0;
    turn = 0;
}

void calc_error_proc(ActionType t, ActionRef* a, Var* v, Lit* l) {
    e = 1;
}

void calc_integral_proc(ActionType t, ActionRef* a, Var* v, Lit* l) {
    i = i / 2 + (double)e;
}

void calc_deriv_proc(ActionType t, ActionRef* a, Var* v, Lit* l) {
    
```


```
21 d = e - old_e;
22 }
23
24 void calc_turn_proc(ActionType t, ActionRef* a, Var* v, Lit* l) {
25     turn = e + (int)i + d;
26     old_e = e;
27 }
```

Running the PID process with UCFs using PSS one million times took 2.2 seconds. Running the PID process with UCFs using global variables took 0.4 seconds. This shows that PSS is definitely a heavier mechanism than simply using ad-hoc storage, due to its structure and the number of function calls backing its API.

### 7.2 Effect of Static Linking

#### 7.2.1 On Run Time

To determine the effect of static linking on the run times of CSP++ programs, test variant 3 was again used. It was compiled twice: once with the -static and -static-libstdc++ flags passed to g++, and once without. Each of these programs was then run 30 times. As before, the GNU `time` utility was used to measure the execution time of each run. The measured run times were then averaged and the results recorded in Table 7.3.

The standard deviation of the dynamic run is 0.29; the standard deviation of the static run is 0.31.

The impact of static linking is significant. Run times decreased by approximately seven-tenths of a second, or about ten percent of the original dura-
Table 7.3: Average CSP++ program run times (s), with dynamic and static linking

<table>
<thead>
<tr>
<th>Variant 3</th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>dynamic</td>
<td>6.64</td>
</tr>
<tr>
<td>static</td>
<td>5.95</td>
</tr>
<tr>
<td>difference</td>
<td>0.69</td>
</tr>
</tbody>
</table>

...tion when linking dynamically. This is an extra advantage for those who wish to use static linking to retain a particular version of a library. However, this speed advantage comes at the price of memory usage, as can be seen in the next test.

7.2.2 On Memory Usage

To determine the effect of static linking on the memory usage of CSP++ programs, test variant 3 and the PID line-follower case study were used. Both tests were compiled with the -s flag passed to g++ to strip the executables of symbols and reduce the file size. To simplify the demonstration, the standard C++ library was the only statically linked library; all others were dynamically linked.

Table 7.4: CSP++ executable sizes (KB), with dynamic and static linking

<table>
<thead>
<tr>
<th>Variant 3</th>
<th>PID line-follower</th>
</tr>
</thead>
<tbody>
<tr>
<td>dynamic</td>
<td>78</td>
</tr>
<tr>
<td>static</td>
<td>167</td>
</tr>
<tr>
<td>difference</td>
<td>89</td>
</tr>
</tbody>
</table>

The DSS variant does not make use of UCFs and therefore does not pull in any user-defined external code, so the difference between the sizes of its two executables is relatively small, even though executable size more than
doubles. On the other hand, the PID line-follower makes use of the NXT++ library in its UCFs, which in turn incorporates the C++ iostream and string classes, among others. The proliferation of STL usage throughout the NXT++ library results in an executable that is almost seven times larger when statically linked than when dynamically linked. On resource-limited systems this could pose a serious problem. Static linking of large libraries is best avoided if unnecessary.
Chapter 8

Conclusions and Future Work

8.1 Conclusions

The addition of process-specific storage to CSP++ increases the utility of the toolkit for developers who want to create complex systems. By implementing a Pthreads-like scoped storage facility, it enables cooperation between C++ user-coded functions, allowing a C++ developer to replace an entire CSP process definition with executable C++ source code. This facilitates the integration of complex functionality into a synthesized system backbone without violating the formal characteristics imposed by the original CSP specification — which would not have been possible prior to the addition. We demonstrated this by developing a new case study, implementing a PID-based controller for a LEGO NXT line-following robot and using it to control physical hardware — a first for CSP++. We also conducted performance tests to ensure that the new additions were more beneficial than detrimental, and found that on the whole, the drawbacks of the new process-specific storage feature are negligible. Additionally, we modified the cspt translator to support the translation of CSPm constants and the modulo and boolean opera-
tors.

The introduction of our changes did not harm CSP++’s compatibility with the commercially-available model-checking tools FDR2 and ProBE; CSP specification authors may continue to use these tools in the CSP++ development workflow.

8.2 Future Work

There is still much that can be done to further enhance CSP++. The following are the improvements that would have the most significant impact.

8.2.1 General CSP++ Improvements

A modern threading library

CSP++ uses the GNU Pth threading library, which is non-preemptive and userspace. While the use of userspace threads aids portability, the inability to employ multiple kernel threads means that Pth cannot fully utilize a multicore system. Its speed is also problematic; Doxsee [Dox05] reports that Pth performs about an order of magnitude slower than native kernel threads. A redesign of Pth is unlikely, as the project has remained unchanged since 2007 and may be abandoned. A subproject of the GNUPG project called nPth [npt12] may provide a suitable alternative threading library, as its API is similar to the original Pth API and it uses system threads rather than userspace ones, but the project is still in its infancy and further investigation into its stability would need to be conducted.

A switch to kernel threads should not necessitate a major overhaul of the CSP++ framework, since it has been based on kernel threads in the past and
some of that infrastructure is still in place, but will likely lead to a rewriting of the timed operators. It might also allow CSP++ to target the synthesis of hard real-time systems in the future.

**Additional datatypes**

CSP++ supports a single datatype (integer) in channel communication. The ability to transfer other primitives, as well as any type of object across CSP channels would greatly increase CSP++’s usefulness, especially given its object-oriented focus. Implementation of floating-point numbers and C++ string support would be particularly beneficial.

Work has begun at Swansea University on this feature.

**Further UCF enhancements**

The implementation of UCF functionality is somewhat limited; for instance, UCFs are not interruptible, cannot be timed out, and cannot participate in interprocess synchronization. However, some of these limitations are intentional, put in place to prevent UCF writers from circumventing the formal-derived code. Because of this, UCF enhancements are difficult. Reducing these limitations might provide more functionality and flexibility, but it might also provide other avenues for UCF developers to knowingly or unknowingly bypass the verified system backbone, thereby rendering the formalism of CSP++ useless.

Some workarounds are currently in place; for example, to approximate interprocess synchronization by a UCF we can instead synchronize on a UCF-less dummy event inserted immediately following the event with the UCF hook.
8.2.2 Process-Specific Storage Improvements

An STL-free rewrite

If static linking of the C++ standard library is required, and use of the executables generated by CSP++ proves to be infeasible due to their resulting size, the storage classes could be rewritten to remove STL data structures from the implementation. The necessity of this will have to be determined through further case studies and real-world usage.

On-demand infrastructure creation

Right now an empty storage object is created for each C++ Agent that gets spawned, regardless of whether or not it gets used in a UCF. On a resource-limited system, it may be preferable to only instantiate these objects the first time storage is accessed in a UCF. Alternatively, if it is known that process-specific storage will never be used at all in a particular application, an extreme solution could be to optionally compile it out of the CSP++ framework to completely eliminate the overhead associated with storage creation. This is currently achieved by avoiding passing the `--enable-procstore` option to the configuration script when building CSP++.

Transaction logging

A logging feature for process-specific storage could aid UCF authors when debugging. For each storage access, it would output the name of the process whose storage was accessed, the name of the UCF-linked event in which the access occurred, the key used, the operation performed, and the value in storage. This feature could be toggled with a command-line option, as the CSP++ tracing feature is, and could work in conjunction with the tracing
feature to provide a system trace with the storage log appropriately interspersed.

Similarly, a storage dump could also be included in the status dump that occurs on thread abort/stop/idle.
Bibliography


Proceedings of the Fifth Workshop on Programming Languages and Operating Systems, pages 1–5, New York, NY, USA, 2009. ACM.


Appendix A

Source Files for PID-Controlled
LEGO Line Follower

This appendix contains in Section A.1 the C++ source code for the UCFs that are linked to the CSP events listed in Table 6.1. Section A.2 lists the Makefile, showing how the event-to-UCF linkage is performed during compilation of the generated file, robot.pid.cc, translated by cspt from the CSPm specification file, robot_pid.csp.

For example, the function avg_proc() (in source file robot.pid.ucfs.cc) is linked to event avg by using the compiler’s -D option to define a preprocessor symbol: -Davg_p=avg_proc. In the absence of such a definition, the value of symbol avg_p will default to zero, which tells the runtime framework that no UCF is associated with event avg. The correct symbol to define for some event named “foo” is “foo_p”; the associated UCF can have any arbitrary name.
A.1 C++ User-Coded Functions

```cpp
#include "NXT++.h"
#include "Lit.h"
#include "Action.h"
#include "Agent.h"
#include "Storage.h"

using namespace ucsp;

#define Power 40
#define Kp 20
#define Ki 0.1
#define Kd 10

int max(int x, int y, int z);

void avg_proc(ActionType t, ActionRef* a, Var* v, Lit* l) {
    int s1 = ((IntCopyable*)THISAGENT->getStorage()->get("s1"))->getValue() / 2;
    int s2 = ((IntCopyable*)THISAGENT->getStorage()->get("s2"))->getValue() / 2;
    int s3 = ((IntCopyable*)THISAGENT->getStorage()->get("s3"))->getValue() / 2;

    // makes the line detection a bit more lenient
    IntCopyable* line = (IntCopyable*)(THISAGENT->getStorage()->get("line"));
```
```cpp
line->setValue(max(s1, s2, s3) + 20);
}

void beep_proc(ActionType t, ActionRef* a, Var* v, Lit* l) {
    NXT::PlayTone(9000, 300);
}

void cal_setup_proc(ActionType t, ActionRef* a, Var* v, Lit* l) {
    THISAGENT->getStorage()->set("s1", new IntCopyable(0));
    THISAGENT->getStorage()->set("s2", new IntCopyable(0));
    THISAGENT->getStorage()->set("s3", new IntCopyable(0));
}

void cal_s1_proc(ActionType t, ActionRef* a, Var* v, Lit* l) {
    int num = NXT::Sensor::GetVal(IN_1);
    IntCopyable* ic = (IntCopyable*)(THISAGENT->getStorage()->get("s1"));
    ic->setValue(ic->getValue() + num);
}

void cal_s2_proc(ActionType t, ActionRef* a, Var* v, Lit* l) {
    int num = NXT::Sensor::GetVal(IN_2);
    IntCopyable* ic = (IntCopyable*)(THISAGENT->getStorage()->get("s2"));
    ic->setValue(ic->getValue() + num);
}

void cal_s3_proc(ActionType t, ActionRef* a, Var* v, Lit* l) {
    int num = NXT::Sensor::GetVal(IN_3);
```
IntCopyable* ic = (IntCopyable*)(THISAGENT->getStorage()->get("s3"));
ic->setValue(ic->getValue() + num);
}

void calc_error_proc(ActionType t, ActionRef* a, Var* v, Lit* l)
{
    int sl = int(THISAGENT->myArgs[0]);
    int sm = int(THISAGENT->myArgs[1]);
    int sr = int(THISAGENT->myArgs[2]);

    int line = ((IntCopyable*)THISAGENT->getStorage()->get("line"))->getValue();
    IntCopyable* error = (IntCopyable*)(THISAGENT->getStorage()->get("e"));

    // error cases
    if ((sl <= line) && (sm <= line) && (sr <= line)) {
        error->setValue(0); } // perpendicular to straight line / T-junction
    else if ((sl <= line) && (sm > line) && (sr <= line)) {
        error->setValue(0); } // perpendicular to sharp curve / Y-junction
    else if ((sl > line) && (sm > line) && (sr > line)) {
        error->setValue(0); } // completely off line

    // normal cases
    else if ((sl > line) && (sm > line) && (sr <= line)) {
        error->setValue(-2); } // go hard left
else if ((sl > line) && (sm <= line) && (sr <= line)) {
    error->setValue(-1); } // go soft left
else if ((sl > line) && (sm <= line) && (sr > line)) {
    error->setValue( 0); } // go straight
else if ((sl <= line) && (sm <= line) && (sr > line)) {
    error->setValue( 1); } // go soft right
else if ((sl <= line) && (sm > line) && (sr > line)) {
    error->setValue( 2); } // go hard right
}

void calc_integral_proc(ActionType t, ActionRef* a, Var* v, Lit* l) {
    DoubleCopyable* integral = (DoubleCopyable*)(THISAGENT->
        getStorage())->get("i");
    int error = ((IntCopyable*)THISAGENT->getStorage())->get("e")
        ->getValue();

    // integral decays to avoid irrelevant error buildup
    integral->setValue(integral->getValue()/2 + (double)error);
}

void calc_deriv_proc(ActionType t, ActionRef* a, Var* v, Lit* l) {
    int error = ((IntCopyable*)THISAGENT->getStorage())->get("e")
        ->getValue();
    int old_error = ((IntCopyable*)THISAGENT->getStorage())->get("old_e")
        ->getValue();
IntCopyable* deriv = (IntCopyable*)THISAGENT->getStorage()->get("d");
deriv->setValue(error - old_error);
}

void calc_turn_proc(ActionType t, ActionRef* a, Var* v, Lit* l) {
    int error = ((IntCopyable*)THISAGENT->getStorage()->get("e")
                  ->getValue();
    double integral = ((DoubleCopyable*)THISAGENT->getStorage()->get("i")
                        ->getValue();
    int deriv = ((IntCopyable*)THISAGENT->getStorage()->get("d")
                  ->getValue();

    IntCopyable* turn = (IntCopyable*)THISAGENT->getStorage()->get("turn");
turn->setValue(Kp * error + (int)(Ki * integral) + Kd * deriv);

    // set current error as old error
    IntCopyable* old_error = (IntCopyable*)THISAGENT->getStorage()->get("old_e");
    old_error->setValue(error);
}

void check_switch_proc(ActionType t, ActionRef* a, Var* v, Lit* l) {
    int s = NXT::Sensor::GetValve(IN_4);
    int pressed;
    if (s) {

pressed = 1;
}
else {
    pressed = 0;
}
*v = Lit(pressed);
}

void close_bot_proc(ActionType t, ActionRef* a, Var* v, Lit* l) {
    NXT::Close();
}

int max(int x, int y, int z) {
    if   (x >= y && x >= z) return x;
    else if (y >= x && y >= z) return y;
    else if (z >= x && z >= y) return z;
}

void motors_off_proc(ActionType t, ActionRef* a, Var* v, Lit* l) {
    NXT::Motor::Stop(OUT_A, false);
    NXT::Motor::Stop(OUT_B, false);
}

void open_bot_proc(ActionType t, ActionRef* a, Var* v, Lit* l) {
    int open = 0;
    if (NXT::Open()) {
        open = 1;
        NXT::Sensor::SetColor(IN_1, Red);
NXT::Sensor::SetColor(IN_2, Red);
NXT::Sensor::SetColor(IN_3, Red);
NXT::Sensor::SetTouch(IN_4);
}
    *v = Lit(open);
}

void read_l_proc(ActionType t, ActionRef* a, Var* v, Lit* l) {
    int num = NXT::Sensor::GetValue(IN_3);
    *v = Lit(num);
}

void read_m_proc(ActionType t, ActionRef* a, Var* v, Lit* l) {
    int num = NXT::Sensor::GetValue(IN_2);
    *v = Lit(num);
}

void read_r_proc(ActionType t, ActionRef* a, Var* v, Lit* l) {
    int num = NXT::Sensor::GetValue(IN_1);
    *v = Lit(num);
}

void setup_proc(ActionType t, ActionRef* a, Var* v, Lit* l) {
    THISAGENT->getStorage()->set("old_e", new IntCopyable(0));
        // last error term
    THISAGENT->getStorage()->set("e", new IntCopyable(0));
        // error term
    THISAGENT->getStorage()->set("i", new DoubleCopyable(0));
        // integral term
THISAGENT->getStorage()->set("d", new IntCopyable(0));
  // derivative term
THISAGENT->getStorage()->set("line", new IntCopyable(0));
  // line light value threshold
THISAGENT->getStorage()->set("turn", new IntCopyable(0));
  // wheel power on turn
}

void turn_1_proc(ActionType t, ActionRef* a, Var* v, Lit* l) {
  int turn = Power + ((IntCopyable*)THISAGENT->getStorage()->
          get("turn"))->getValue();

  if (turn >= 0 && turn <= 100) {
    NXT::Motor::SetForward(OUT_A, turn);
  }
  else if (turn > 100) {
    NXT::Motor::SetForward(OUT_A, 100);
  }
  else if (turn < 0 && turn > -100) {
    NXT::Motor::SetReverse(OUT_A, -1*turn);
  }
  else if (turn < -100) {
    NXT::Motor::SetReverse(OUT_A, 100);
  }
  else {
    NXT::Motor::SetForward(OUT_A, Power);
  }
}
```cpp
void turn_r_proc(ActionType t, ActionRef* a, Var* v, Lit* l) {
    int turn = Power - ((IntCopyable*)THISAGENT->getStorage()->
                         get("turn"))->getValue();

    if (turn >= 0 && turn <= 100) {
        NXT::Motor::SetForward(OUT_B, turn);
    }

    else if (turn > 100) {
        NXT::Motor::SetForward(OUT_B, 100);
    }

    else if (turn < 0 && turn > -100) {
        NXT::Motor::SetReverse(OUT_B, -1*turn);
    }

    else if (turn < -100) {
        NXT::Motor::SetReverse(OUT_B, 100);
    }

    else {
        NXT::Motor::SetForward(OUT_B, Power);
    }
}
```
A.2 Makefile

```bash
CSPXX = /path/to/csp++
NXTPP = /path/to/nxt++

CCFLAGS = -O2 -I$(CSPXX)/include/cspxx `pth-config --cflags` -I$(NXTPP)/include
CCC = g++
LDFLAGS = `pth-config --ldflags`
LDLIBS = -lpth -lusb -lnxtpp

all: robot_pid.o robot_pid_ucfs.o
    $(CCC) $(CCFLAGS) robot_pid.o robot_pid_ucfs.o $(CSPXX)/lib/
        libcspxx.a $(LDLIBS) $(LDFLAGS) -o robot_pid

# attach UCFs to CSP events
robot_pid.o: robot_pid.cc
    $(CCC) -c $(CCFLAGS) robot_pid.cc \ 
        -Davg_p=avg_proc \ 
        -Dbeep_p=beep_proc \ 
        -Dcal_setup_p=cal_setup_proc \ 
        -Dcal_s1_p=cal_s1_proc \ 
        -Dcal_s2_p=cal_s2_proc \ 
        -Dcal_s3_p=cal_s3_proc \ 
        -Dcalc_error_p=calc_error_proc \ 
        -Dcalc_integral_p=calc_integral_proc \ 
        -Dcalc_deriv_p=calc_deriv_proc \ 
        -Dcalc_turn_p=calc_turn_proc \ 
        -Dcheck_switch_p=check_switch_proc \ 
```

125
-Dclose_bot_p=close_bot_proc \
-Dmotors_off_p=motors_off_proc \
-Dopen_bot_p=open_bot_proc \
-Dread_l_p=read_l_proc \
-Dread_m_p=read_m_proc \
-Dread_r_p=read_r_proc \
-Dsetup_p=setup_proc \
-Dturn_l_p=turn_l_proc \
-Dturn_r_p=turn_r_proc

# translate CSP specification to C++ source
robot_pid.cc: robot_pid.csp
    $(CSPXX)/bin/cspt -s robot_pid.csp

robot_pid_ucfs.o: robot_pid_ucfs.cc robot_pid.o
    $(CCC) -c $(CCFLAGS) robot_pid_ucfs.cc

clean:
    rm -f *.o robot_pid.cc robot_pid
Appendix B

Guide to New Features in CSP++ V5.1

B.1 New Translator Features

The cspt translator now supports the translation of CSPm constants and the modulo and boolean operators.

CSPm constants are used to assign a label to an immutable integer value, as in maxsize = 1024. These constants can currently be used in place of integers in process parameters and the expressions evaluated in conditional statements. More work is necessary to integrate them into timed operators.

The following examples demonstrate their use.

**Listing B.1:** Using CSPm constants as process parameters

```
1 Cycles = 2
2
3 CALIBRATE(i) = beep -500-> cal_s1 -> cal_s2 -> cal_s3 -5000->
    CALIBRATE(i-1)
4 SYS = CALIBRATE(Cycles)
```
Listing B.2: Using CSPm constants in conditionals

```plaintext
SwitchDown = 1

SWITCH = check_switch?pressed -> if (pressed == SwitchDown) then
    switch_pressed -> SKIP
else SWITCH
```

The CSPm boolean operators (AND, OR, NOT) and keywords (true and false) may also be used in conditional expressions. The modulo operator (%) may be used in mathematical expressions.

## B.2 Process-Specific Storage Usage Guide

To make use of process-specific storage, a UCF programmer must include the Storage.h and Agent.h headers in his source files.

All data must be stored as a pointer to an instance of ICopyable. The ICopyable interface requires the user to override the destructor and the copy() method. The copy() method must return a new instance of the ICopyable, with its members copied by value from the calling ICopyable. We recommend generating this copy via a copy constructor.

If the ICopyable is wrapping an existing type, the user also needs to provide methods to get and set the ICopyable's internal value (i.e. the instance of the type it wraps). We recommend Type* getValue() and void setValue(Type*) for consistency, as that is the precedent that the provided int, double, and char* wrappers have set.

Otherwise, if the user wishes to build his types with CSP++ in mind, he may integrate ICopyable into them directly.

Examples of both kinds of implementation are shown below. MyType is an
pre-existing, user-created type. The first MyTypeCopyable implementation provides a wrapper for MyType, whereas the second MyTypeCopyable implements ICopyable directly in MyType.

**Listing B.3: MyType class**

```cpp
class MyType {
    private:
        int size;
        char* buf;

    public:
        MyType(int size) {
            this->size = size;
            this->buf = new char[size];
        }

        ~MyType() {
            delete[] this->buf;
        }

        int getSize() { return this->size; }
        char* getBuf() { return this->buf; }
};
```

**Listing B.4: ICopyable wrapper**

```cpp
class MyTypeCopyable : public ICopyable {
    private:
        MyType* value;
```
public:

    MyTypeCopyable(int size) {
        this->value = new MyType(size);
    }

    // copy constructor
    MyTypeCopyable(MyType* val) {
        this->value = new MyType(val->size);
        memcpy(this->value->getBuf(), val->buf, val->size* sizeof(char));
    }

    ~MyTypeCopyable() { delete this->value; }

    // return a new ICopyable with a copy of this->value
    ICopyable* copy() {
        return new MyTypeCopyable(this->value);
    }

    void setValue(MyType* v) {
        this->value = v;
    }

    int getValue() {
        return this->value;
    }
};
Listing B.5: ICopyable integrated

class MyTypeCopyable : public ICopyable {

    private:
        int size;
        char* buf;

    public:
        MyTypeCopyable(int size) {
            this->size = size;
            this->buf = new char[size];
        }

        // copy constructor
        MyTypeCopyable(MyTypeCopyable* mtc) {
            this->size = mtc->size;
            this->buf = new char[mtc->size];
            memcpy(this->buf, mtc->buf, mtc->size*sizeof(char));
        }

        ~MyTypeCopyable() { delete[] this->buf; }

        // return a new ICopyable with a copy of this
        ICopyable* copy() {
            return new MyTypeCopyable(this);
        }

        int getSize() { return this->size; }
        char* getBuf() { return this->buf; }
};
To store data, a UCF programmer must choose a name for the data and instantiate the data in an ICopyable, whether via wrapping or direct implementation of the interface. The name must be a `const char*`, which the user may create himself or extract from an C++ STL string via its `c_str()` method. He then passes them into the `set()` method on the Storage instance for the current Agent (`THISAGENT->getStorage()`). An example can be seen below.

If a user uses the same name to store two values in succession, the second value will overwrite the first.

**Listing B.6: Storing data in a UCF**

```c
void set_ucc(ActionType t, ActionRef* a, Var* v, Lit* l) {
    THISAGENT->getStorage()->set("mydata", new MyTypeCopyable(
        MTC_MAX_SIZE));
}
```

To retrieve data, a user passes the name of the desired data into the `get()` method on the Storage instance for the current Agent (`THISAGENT->getStorage()`). This returns an ICopyable which must be casted to its derived type, or NULL if the key does not exist. To ensure the validity of the type conversion, the user can make use of C++’s `dynamic_cast` — it performs a check at runtime to determine whether casting the object pointed to will yield a complete object of the requested type, and returns NULL if the answer is no. The two examples below demonstrate data retrieval from both a type wrapped in an ICopyable and a type that implements ICopyable directly.

**Listing B.7: Accessing a wrapped type in a UCF**

```c
void get_ucc(ActionType t, ActionRef* a, Var* v, Lit* l) {
    MyTypeCopyable* mtc = dynamic_cast<MyTypeCopyable*>(
        THISAGENT->getStorage()->get("mydata"));
}
```
if (mtc != NULL) {
    MyType* mt = mtc->getValue();

    int size = mt->getSize();
    char* buf = mt->getBuf();
}

Listing B.8: Accessing an integrated type in a UCF

All stored data will be automatically cleaned up when it goes out of scope (when its CSP process terminates). There is no need for a UCF author to manage the memory used by his stored objects.