REENGINEERING CSP++ TO CONFORM WITH CSPM VERIFICATION TOOLS

A Thesis

Presented to

The Faculty of Graduate Studies

of

The University of Guelph

by

STEPHEN DOXSEE

In partial fulfilment of requirements

for the degree of

Master of Science

August, 2005

ABSTRACT

REENGINEERING CSP++ TO CONFORM WITH CSPM VERIFICATION TOOLS

Stephen Doxsee
University of Guelph, 2005

Advisor:
Professor W. B. Gardner

The formal process algebra, Communicating Sequential Processes (CSP), has been widely used for the modeling and verification of concurrent and real-time systems. The tool, CSP++, was developed to apply a technique called “selective formalism” that makes formal CSP specifications executable via automatic code generation, and extensible using C++ user-coded functions. However, besides CSP++ being a proof-of-concept tool in need of further consolidation, it lacked one of the key benefits of the selective formalism design flow, verification, because it accepted a dialect of CSP without commercial tool support.

In this work, we present a reengineered CSP++ to conform with the CSPm dialect of CSP and the commercial tools for it. The CSP++ translator and framework are extensively reengineered to improve the usefulness and power of the tool. The new CSP++ is demonstrated with a new Automated Teller case study that applies the selective formalism design flow. Changes in the performance of the tool are measured by comparing with previous versions and Rational Rose RealTime.

Acknowledgements

Much gratitude is owed to many people, whose various means of support made this research possible:

- First and Foremost to my supervisor, Dr. Bill Gardner, for modeling and fostering excellence, hard work, and integrity; for giving me opportunities to grow and be stretched; for his hospitality and kindness in regularly having me at his home; but most of all for his great friendship that began years before I even started at Guelph and has continued to grow deeper.
- To my parents for their sacrifices of love, food, and vehicle.
- To Ben Hanna, John Carter, and Ming Xu for being my companions "in the trenches" of masters studies and CSP research.
- To Dr. Gary Grewal for his friendship and thorough examination of this research.
- To Edna Mumford and Debra Byart for their help and kindness.
- To Dr. Ariel and Prof. Sutcliffe for their exemplary lives and care in my undergraduate education at Trinity Western University.
- And finally, to other family and friends whose prayers and encouragement kept me pressing to the finish.

Table of Contents

Page

Acknowledgements i

Table of Contents ii

List of Figures vii

List of Tables viii

Chapter 1 Introduction 1

1.1 Problem Definition and Motivation 3

1.2 Research Approach and Contributions 5

1.3 Thesis Outline 6

Chapter 2 Background and Related Work 8

2.1 CSP Overview 8

2.1.1 Processes 10

2.1.2 Composition and Synchronization 11

2.1.3 Events 12

2.1.4 Communication 13

2.1.5 Other Operators 16

2.1.6 Sections of a CSPm Specification 17

2.2 Terminology 19

2.3 Related Work 21

2.3.1 Formal Methods 22
2.3.2 Special-Purpose Languages 23
2.3.3 Libraries for General-Purpose Languages 24
2.3.4 CSP-Based Synthesis Tools 25
2.3.5 Application Areas for CSP-Based Systems 28
Chapter 3 Reengineering CSP++ for CSPm 29
3.1 Theoretical Issues 30
3.1.1 Data Types Supported 31
3.1.2 Events Supported 34
3.1.3 Processes Supported 38
3.1.4 Composition 39
3.1.5 Communication 40
3.1.6 Operators Supported 45
3.1.7 Other Supported Features 47
3.1.8 UCF Integration 47
3.1.9 Unsupported Features 48
3.2 Translator Changes 50
3.2.1 Reengineering the Front-End of CSP++ 51
3.2.2 Overview of cspt Translator 54

3.2.3 Simple Changes 61

3.2.4 Channel I/O 62

3.2.5 Data Types 63

3.3 Framework Changes 63

3.3.1 Parametric Changes 64

3.3.2 Subscripted Channels 64

3.3.3 Event Sets 65

3.3.4 Multilevel Synchronization 67

3.4 Summary of Restrictions and Supported Syntax 73

3.5 Restrictions and Limitations 73

Chapter 4 Automated Teller Case Study 77

4.1 CSP in the Design Phase 79

4.2 Writing the CSPm Specification 82

4.2.1 Deriving CSPm from Statecharts 82

4.2.2 Handling Data and Variables 83

4.2.3 Choosing an Approach for Modeling the ATM in CSPm 85

4.3 Verification 87

4.3.1 Basics of Verification 89

4.3.2 Trace Refinement 89

4.3.3 Failures Refinement 90

4.3.4 State Space 91

4.4 Synthesizing C++ and Integrating UCFs 92

Chapter 5 Performance Metrics 98

5.1 The Effect of Pth 100

5.2 Static vs. Dynamic Linking in V3.0 102

5.3 The Effect of Specification Structure 103

5.4 The Effect of CSPm Modifications 105

5.4.1 DSS Performance 105

5.4.2 ATM Performance 106

5.5 The Effect of isTop Feature 108

5.6 Memory Estimates 110

Chapter 6 Conclusions and Future Directions 112

6.1 Conclusions 112

6.1.1 A More Useful Tool 112

6.1.2 A More Powerful Tool 113

6.1.3 Competitive Performance 113

6.2 Future Directions 114

6.2.1 Changing the Thread Model 115

6.2.2 Porting CSP++ to Embedded Systems 116

6.2.3 Enhancing UCF Integration 117

6.2.4 Implementing More CSPm Features 117

6.2.5 Supporting More Data Types 117

6.2.6 Finding a Way to Model Time 118

6.3 Status and Availability of CSP++ 119

References 120

Appendix A ATM Statecharts 124

Appendix B ATM Code 127

B.1 CSPm Specification for ATM 129

B.2 User-Coded Functions 133

B.3 Bank Simulation 140

Appendix C Training CSP++ Personnel 141

List of Figures

Page

Figure 1 Support for CSPm and Hoare's CSP. 41

Figure 2 Multilevel Synchronization and Communication..... 71

Figure 3 ATM Use Case Diagram 79

Figure 4 Correspondence between Statecharts and CSPm 86

Figure 5 Environment Stack of Specification Demonstrating isTop Feature 109

Figure 6 Statechart for Overall ATM..... 124

Figure 7 Statechart for One Session..... 125

Figure 8 Statechart for One Transaction..... 126

List of Tables

Page

Table 1 General Terms 20

Table 2 CSPm vs. csp12 Terms 21

Table 3 Comparison of CSP++ and Raju for supported CSPm features 26

Table 4 BNF syntax with corresponding parse node classes 56

Table 5 Restrictions in current CSP++ 74

Table 6 Locations of Limitations 75

Table 7 Total Time (Seconds) for Disk Accesses with Pth and LinuxThreads, V3.0 101

Table 8 Total Time (Seconds) for 20000 Disk Accesses, V3.0 102

Table 9 Time for 10000 ATM transactions (Structure Modifications) 104

Table 10 Time for 20000 simulated disk accesses 106

Table 11 Time for 10000 ATM transactions 107

Table 12 Time for 20000 disk accesses (isTop feature) 109

Table 13 CSP++ Object File Sizes 110

Table 14 UCFs for the ATM 133

Table 15 CSP Seminar Outline 142

Chapter 1

Introduction

When a spacecraft is sent into outer space, its developers cannot afford to send it out with faulty software. Systems such as these that must work correctly, likely have a formal element to their development. The use of formal methods enables systems to be reasoned about mathematically to ensure the system is free of problematic properties such as deadlock, where a system can no longer make progress because every process is waiting on some other process.

However, there are at least a few large drawbacks to the way formal methods are often used in system development. First, although formal methods may be used in the initial design of the system, the formal specification may become out-of-sync with the code base as the system evolves over time. This makes the original specification an artifact of the system design that has less and less relevance to the developing system. Second, hand-translation of formal specifications to a software implementation is an error-prone and time-consuming process. Third, it is often impractical to specify the entire system formally, and many components of a candidate system for design by formal methods can be implemented more easily using conventional programming languages.

Software synthesis (that is, automatic code generation via design automation tools) from formal specifications is one approach to solving the first two problems. Synthesis ensures that the code and specification continue to correspond with each other, since the former is generated directly from the latter. Furthermore, automatic software synthesis is quick and it eliminates human error in translation, provided that the synthesis process is

correct. Finally, the third drawback can be solved by specifying only the critical portions of the system formally and leaving the rest to be implemented in a conventional programming language. In general, “critical” portions include the system’s control flow, where concurrency pitfalls such as deadlock may appear. Certainly, anything involving human safety would be “critical.” In contrast, portions that may not warrant formal treatment include straightforward data processing and I/O operations.

There has been an effort over a number of years to develop a tool named CSP++ [Gardner 2000, Gardner 2003], created by W. B. Gardner as part of his Ph.D. thesis at the University of Victoria, that makes a formal algebra, Communicating Sequential Processes (CSP) [Hoare, C. A. R. 1985], both directly *executable* via software synthesis and *extensible* using C++ User-Coded Functions (UCFs). The technique, later dubbed “selective formalism” [Gardner 2003], allows system designers to select which portions of the system are important to be specified formally and which portions can be left to conventional language programmers. CSP++ applied selective formalism by combining synthesized code from CSP specifications with C++ user-coded functions via an Object-Oriented Application Framework (OOAF) that executes the CSP semantics. The executed events can optionally be displayed through a tracing option as can the memory, actions, and errors by setting their respective flags. Selective formalism via CSP++ is a “best of both worlds” approach for developing systems that profits from formal verification as well as utilizing the existing skills of conventional language programmers.

CSP++ successfully demonstrated selective formalism as a “proof of concept” tool. However, for CSP++ to become a viable software engineering tool, it must overcome certain problems and move from its current status to maturity. In the rest of this chapter

we will introduce the motivation and approach for reengineering CSP++ and explain the organization of this thesis.

1.1 Problem Definition and Motivation

The most pressing concern with CSP++ is that the dialect of CSP it uses is extremely local, thus limiting its usefulness. The dialect, **csp12** [Cheng 1994], was developed by M. H. M. Cheng at the University of Victoria, for an in-house verification tool. It is not widely adopted and does not have nearly the support of the machine-readable CSP dialect called **CSPm**. Case studies developed for CSP++ have had to be hand-translated to CSPm in order to have the support of the most popular commercial CSP tools, Failures-Divergence Refinement 2 (FDR2) for verification, Process Behaviour Explorer (ProBE) for simulation, and Checker for type checking, all from Formal Systems [Formal Systems]. FDR2 is a refinement checker that allows the properties of a system modeled in CSPm to be verified. Such a tool facilitates the exposure of deadlock, liveness, nondeterminism, and other properties lurking in the system. The use of FDR2 verification can lead to increased confidence in the soundness of the system model by detecting undesirable properties and enabling the specification writer to correct the problems. Ideally, a CSP specification should be able to be directly verified, simulated, and synthesized from the same dialect. Without a common dialect, there is the inevitable painful task of keeping the verifiable and executable versions of the specification in sync.

Another concern with CSP++ is that it is missing some desirable features that are part of the CSPm dialect. As a result, CSP++'s capabilities for developing software systems are limited. As case studies were developed for CSP++, it became clear that not only is the input syntax limiting CSP++'s usefulness but the lack of support in the OOAF for

some features were limiting its power. Some of those limitations, including multilevel synchronization and compound events, will be discussed in section 3.1.4 and section 3.1.5.

Finally, a tool that aspires to industry adoption needs to show competitive performance. Prior to this research, performance measurements on CSP++ were extremely cursory. Furthermore, over its lifespan, the tool has undergone changes that could affect its performance, especially in the area of its underlying thread model (see section 5.1).

CSP++ began by using the AT&T USL (Unix Systems Laboratory) task library—non-preemptible, non-prioritized coroutines. By 1999, the AT&T task library was essentially obsolete, so CSP++ was changed to use the preemptible, kernel-scheduled LinuxThreads implementation of POSIX threads that came as part of the Red Hat Linux distribution. This version of CSP++ has been demonstrated to work in the development of a Disk Server Subsystem (DSS) case study resulting in C++ code that ran at speeds comparable to another code generation tool, Rational Rose RealTime (RRRT, formerly known as ObjecTime, recently rebranded as IBM Rational Technical Developer).

Because the original CSP++ design was based on non-preemptible threads, there was always a suspicion that the move to preemptible threads was done too hastily and without complete identification of critical sections in the framework code. Therefore, the preemptible threads version was not regarded as reliable, and a search was made for another non-preemptible threads package. In 2004, the portable non-preemptible, user-space scheduled implementation of POSIX threads, GNU Pth, was adopted without much change to the OOAF. It increased the portability of CSP++ and continues to be CSP++'s

threading model for the time being. Changes such as these need to be scrutinized to discover their effect on CSP++'s performance to ensure that CSP++ continues to execute at speeds comparable to RRRT.

In the light of the problems facing CSP++, it clearly had to be reengineered for CSPm specifications to continue to have research potential or relevancy to the software development industry. We decided to reengineer CSP++ to conform with CSPm verification tools in order to create a more useful and powerful tool with continued competitive performance, and to carry out extensive meaningful performance measurements.

1.2 Research Approach and Contributions

There were a number of steps that had to be taken to achieve the research goals of improving the usefulness and power of CSP++ while keeping it at a competitive performance level.

CSP++ is composed of a front-end translator, called **cspt**, and the back-end OOAF that the translated code runs with. Both were reengineered to conform with CSPm. This enabled specifications to be both directly verifiable with FDR2 and directly synthesizable via CSP++ without hand translation between dialects, as would previously have been required. A translator that accepts the CSPm input syntax significantly increases the usefulness of CSP++ and facilitates the development of case studies. With that said, CSPm has some features that had no analog in the former dialect of csp12 and for which no back-end support mechanism in the CSP++ OOAF has been provided. Back-end support was targeted for CSPm features that are amenable and beneficial for synthesis,

thus restricting support to a synthesizable subset of CSPm. In particular, it was not attempted to synthesize the nondeterministic constructs of CSPm as they would result in arbitrary choices without control of the system or its environment. Wherever the semantics of csp12 are significantly different from CSPm, CSP++ behaves with the semantics of the supported subset of CSPm.

Clearly, the single DSS case study is insufficient to give us confidence in the usefulness, power, and performance of CSP++. CSP++ needed to be tested with larger and more feature-rich case studies to help explore its scalability and capability. Thus, we demonstrate the usefulness of the reengineered CSP++ tool with a new Automated Teller (ATM) case study complete with examples of verification, and performance metrics and analysis.

Detailed attention was given to the performance of CSP++ by comparing the new CSP++ for CSPm with Rational Rose RealTime and with previous versions of CSP++. Furthermore, questions as to how the CSP specification structure affects the performance of the system were examined by timing the synthesized ATM case study with several different specification variations. The effects of changing the underlying thread model since the last performance metrics were taken were thoroughly investigated.

1.3 Thesis Outline

This thesis is organized as follows: Chapter 2 provides the background for the thesis, including terminology and tables, as well as an overview of previous CSP++ work and other related work. In Chapter 3, we discuss the theoretical issues that had to be considered in the reengineering of CSP++ as well as the detailed technical changes that

were accomplished in phases. Chapter 4 presents the new ATM case study including the design flow, issues involved, and lessons learned. In Chapter 5, we examine the performance of the reengineered CSP++ by comparing the execution times of various benchmarks running on different versions of CSP++ as well as Rational Rose RealTime. Chapter 6 gives the conclusions and possible future directions of CSP++ research. Following the references, there are a number of appendices with case study requirements (Appendix A), case study CSPm and C++ source code (Appendix B), and the description of a seminar series developed by the author to train CSP++ personnel (Appendix C).

Chapter 2

Background and Related Work

Chapter 2 provides the background information necessary to understand the CSP language and the terminology used in this thesis. After an explanation of the previous work on CSP++ a discussion of the related work that situates CSP++ among other attempts to apply formal methods to systems engineering is provided.

2.1 CSP Overview

This section provides a brief overview of CSP and CSP++. CSP is presented with the aim of preparing the reader with enough CSP background to understand simple applications using CSP++. For more information and tutorials on CSP, [Concurrent and Real-time Systems: the CSP Approach] provides helpful materials.

To minimize confusion, a clear distinction in terminology must be made. Communicating Sequential Processes (CSP) was invented by Tony Hoare [Hoare, C. A. R. 1985]. Since there is no international CSP standard, there are many interpretations of CSP with various features and operators added to enhance CSP for a specific use. Therefore, in our work, the term CSP will refer to the interpretation of Schneider [Schneider 2000], and **CSPm** will refer to the specific machine-readable dialect of CSP used by the Formal Systems tools [Schneider 2000, FDR2 User Manual]. CSPm will be the notation used throughout the thesis for all examples of CSP. Finally, **CSP++**, as well as referring to the tool, will be the term used to refer to the synthesizable subset of CSPm

supported by the new reengineered CSP++ unless explicitly qualified by, for instance, “the csp12 version of CSP++” or “the old/former CSP++.”

CSP’s great strength is that it defines a formal semantics for interprocess synchronization and communication, which are often the most error-prone and troublesome areas of concurrent systems. If not properly defined, synchronization and communication can lead to undesirable situations like deadlock. Many software systems suffer from issues arising from concurrency that could be avoided if first modeled in CSP. The formal semantics of CSP enable specifications to be verified to find and eliminate these dangerous properties.

The CSP formalism contains a small number of fundamental elements that are used to write CSP specifications. The core of a CSP specification is made up of a number of *process* definitions. For example, a process might represent the actions of the operator of an ATM. Processes engage in sequences of named *events* such as turning an ATM on. The record of the sequence of events resulting from a process execution is called a *trace*. The set of all a process’s events constitutes the *alphabet* of the process. An operator’s alphabet would be turning the ATM on or off as well as setting the amount of money the machine holds.

Here is a simple example of a two processes in parallel where one, P(5), requests the other, SQUARE, to calculate the square of the number 5 and send back the result:

```
P(n) = square!n -> result?r -> SKIP
SQUARE = square?x -> result!x*x -> SQUARE
SYSTEM = P(5) [|{|square, result|}|] SQUARE
```

The CSPm constructs in this example will be explained below, including how processes are defined and composed into complex systems, the format of events, the operators for

expressing interprocess communication and choice, and the components of a typical CSPm specification.

2.1.1 Processes

In CSP++, processes are defined in terms of the events they engage in and in terms of other processes. Simple processes are defined by a process name, an '=' sign, and a sequence of events separated by prefix '->' operators, ending with another process's name. Below are a several simple process definitions:

```
P = a -> b -> c -> SKIP
Q = r -> a -> s -> T
T = d -> SKIP
```

Process P executes the events 'a', 'b', and 'c' in sequence followed by the termination process, SKIP. CSP defines two built-in processes for termination—SKIP for normal termination and STOP for abnormal deadlocked termination. Q's trace would be <r,a,s,d> since it executes 'd' after process Q continues as T.

Processes can also be defined in terms of themselves, as can be seen in the following example:

```
U = e -> f -> U
```

Process U executes 'e' then 'f' and then returns back to the beginning of process U. Tail recursion, as the above example illustrates, is common in specifications and is implemented efficiently as a special case in CSP++ using looping.

CSP also defines parameterized processes as a way for data to be passed from one process to another. In the squaring system above, P(5) invokes the parameterized process

P(n), where n serves as the formal parameter replaced by 5. Another example of a parameterized process is given in section 2.1.4.

2.1.2 Composition and Synchronization

CSP provides several means of combining processes. Using CSP to model the composition of processes is useful for software engineering because systems are often made up of multiple threads or processes that interact with each other in various ways. In CSP++, CSP processes are mapped to individual threads by the OOAF. CSP++ allows three kinds of composition:

1) Sequential: P;Q
2) Synchronized: P [|{a}|] Q
3) Independent: P ||| Q

Sequential composition has the effect of executing P until it terminates normally, then immediately resuming execution as Q. So, P;Q (using the P and Q defined in the previous section) would yield the following trace: <a,b,c,r,a,s,d>.

Synchronized composition (also called interface parallel) allows P and Q to run concurrently and independently *until* they wish to execute an event in their common alphabet that is listed in the synchronization set (i.e., ‘a’ in the example above). Synchronized composition performs barrier-style synchronization where if one process is ready to synchronize on an event, it must wait for all processes to be ready as well. Two possible traces of P [|{a}|] Q would be <r,a,b,c,s,d> and <r,a,s,b,c,d>. P and Q perform ‘a’ *only once* by both processes together (indicated by an underlined ‘a’ in the trace) and then resume independent execution until the next instance of ‘a’. Although the laws for

CSP specifications dictate that P will execute its events in the order they were defined (i.e., <a,b,c>), the laws also permit the overall execution order of concurrent processes to vary from run to run since CSP defines loose execution semantics. Notice that in the traces of the two synchronized processes, ‘b’ never happens before ‘a’ and ‘c’ never happens before ‘b’ since P specifies that they occur in the order ‘a’ then ‘b’ then ‘c’. There are actually six different possible traces in total for P [|{a}|] Q.

Independent composition (also called interleaving parallel) allows P and Q to execute concurrently without any possible synchronization between them. One resulting trace for P ||| Q would be <a,b,c,r,a,s,d>.

CSP and CSPm also define other composition operators. The alphabetized parallel operator—(P [A||B] Q, where A and B are sets of events—behaves identically to the interface parallel operator except that the synchronization set is calculated as the set intersection of A and B. Both CSP and CSPm define replicated composition operators to allow specification writers to write many synchronizations with less verbiage. All these extra composition operators are for convenience and are not necessary for CSP++.

2.1.3 Events

So far, we have seen only simple events, i.e., events without multiple components. CSP also allows *compound events* made up of multiple components separated by dots. The first component of any event (simple or compound) is what CSPm calls a “channel name”. Following the “channel name” are any number of dot-delimited values. CSP++ currently supports integer values, but CSPm provides for additional data types.

Although the nomenclature of “channel” suggests the communication of data, compound events need not actually transfer data between processes. The values appended to the “channel name” may instead (or additionally) constitute subscripts, so as to define more detailed events while remaining in the same channel family of events.

For example, ‘start.3.22’ could represent an event $start_{3,22}$ meaning the “start” of a marathon runner in age category 3 and wearing the number 22. The same “start” channel name could be used to represent a different marathon runner as well (e.g., ‘start.3.14’).

Subscripts themselves do not communicate data but they may be combined with data values to perform interprocess I/O communication (described in the next section). From CSPm `channel` declarations alone, it is not obvious how to distinguish between subscripts and data values when one sees a particular compound event, or even between subscripted channels used solely for synchronization and channels (subscripted or otherwise) used for I/O. These distinctions should be made clear by the designer of the specification by inserting sufficient comments. Compound events for data communication are discussed in the following section.

2.1.4 Communication

We have already seen that processes can synchronize on a common event, whether simple or compound. In addition, data can be communicated from one process to another by using the CSPm output ‘!’ and input ‘?’ operators on the same “channel name”. If the `channel` is subscripted, the subscripts must match as well as the name for communication to take place. So, for example, ‘c.1?x’ would synchronize with ‘c.1!2’ to communicate

the data value 2, because the channel name ‘c’ and the subscript ‘1’ match, and the data values and variables are compatible in number.

According to Hoare’s original convention [Hoare, C. A. R. 1985], “channels” (in the I/O sense) can be considered a kind of structural component that designers use as data “pipes” between pairs of processes. As such, a given (subscripted) channel has four properties. Its communication is:

1) between a particular pair of processes
2) in only one direction
3) synchronous
4) non-buffered

Non-buffered and synchronous communication means that a communicating process blocks until a value is transferred to the receiver, thus communication in CSP always implies synchronization.

CSPm takes a less restrictive view of the first two communication rules by allowing communication between any number of processes and in more than one direction. CSPm communication becomes, in effect, a pattern-matching operation, where all processes that offer the same combination of channel name, subscripts, and output values (these are effectively the outputting processes) can synchronize with all inputting processes that do the same. The ‘?*variable*’ parts of input events act as wildcards, with positionally corresponding data values being bound to those variables. This interpretation of “I/O” results in the same traces as Hoare’s channels, but permits more flexible communications such as broadcasting (i.e., relaxing the first property). CSP++ is also relaxed on the first

two communication rules as it allows the broadcasting of data from one output process to multiple input processes, and its channels can communicate in either direction.

In CSPm, communication happens over channels via a “communication field.” A communication field follows a channel name (and any optional subscripts), begins with an input or output operator (i.e., ‘?’ or ‘!’ respectively), and is completed by a number of variables (for input) or a number of values (for output). An I/O operation becomes an event when enough values have been supplied to bind all the free variables [FDR2 User Manual].

Processes communicate data through channels during synchronization. If the processes are not specified to synchronize, no data can be communicated. Therefore, a synchronized process composition has to specify a set containing every possible communication event, i.e., every relevant combination of channel name, subscripts, and data values. Since the set may well be infinite, some special notation is needed. CSPm provides the “set closure” notation {|..|} as a simple way to refer to all the possible events for a given channel name without listing them exhaustively. For example, if channel ‘c’ was declared to permit integer data values from 3 to 5, then {|c|} would be equivalent to the set {c.3,c.4.c.5}.

Here is an example of how processes communicate.

```
P = ... -> c!5 -> ...
Q = ... -> c?x -> ...
R = P [|{|c|}|] Q
```

R specifies that P and Q will synchronize on any event that begins with a ‘c’ (due to the set closure notation around ‘c’). As P and Q synchronize on the compound event ‘c.5’ and communicate over channel ‘c’, Q’s free local variable ‘x’ is bound to the value 5

supplied by process P. In Q, the value of ‘x’ will be available for the remainder of the process and may even be passed to a subsequent parameterized process as in the following example.

```
Q = c?x -> S(x)
S(x) = d!x -> SKIP
```

Above, we see that once the value ‘x’ is received by channel ‘c’ in the process Q, ‘x’ can be passed as a parameter to the process S(x) to be output along channel ‘d’, resulting in the event ‘d.5’.

2.1.5 Other Operators

One key feature of CSPm is its ability to specify external (deterministic) choice using the ‘[]’ operator. The external choice operator is illustrated below.

```
R = a -> P [] b -> Q
S = E [|{a,b}|] R
```

In the above system S, process R offers a choice to the environment process, E, between performing an ‘a’ or a ‘b’. If E wants to do ‘a’, R resolves the choice as an ‘a’ and then continues on as the process P. If instead E offers a ‘b’, then R resolves the choice as a ‘b’ and then continues on as the process Q. In either case, only one ‘a’ or ‘b’ is executed and entered in the trace of S.

CSP++ defines other operators including hiding and renaming as well as arithmetic and if/then/else conditional statements. A full table of CSPm operators and the CSP++ supported subset appears in Table 3 of section 2.3.4.1.

2.1.6 Sections of a CSPm Specification

A CSPm specification as input to FDR2 for verification is typically composed of three different sections of definitions: declarations, process specifications, and verification assertions. Only the process specifications are strictly necessary for synthesis in CSP++. However, in order for the new CSP++ to accept CSPm specifications directly, we allow but ignore declarations and verification assertions. The three sections of a CSPm specification are described below.

1) Declarations

   In order for the Formal Systems CSPm tools to know the types of events and data types that are being used in the specification, they require declarations. This section declares each channel that is used in the specification and specifies the ranges of values that can be used for the data parts of these channels. The following are examples of a `channel` and `nametype` declarations.

```
channel date: Month.Day
nametype Month = {1..12}
nametype Day = {1..31}
```

   With the above declaration, a specification writer could use an event like ‘date.12.25’.

2) Process Specifications

   The bulk of the specification is written in the process specification section. This is where processes are defined to form a hierarchical specification of the system using the CSPm elements introduced earlier in this section.

3) Verification Assertions

In the verification section, propositions can be made about processes in the specification.

FDR2 supports both “trace refinement” and “failures refinement” [Schneider 2000] for thorough verification of CSPm specifications. Trace refinement (or a safety specification) is written using the trace refinement operator, ‘[T=’, as in:

```
assert P [T= Q
```

where the assertion will be true if the traces of Q are a subset of the traces of P[1]. The safety specification above could be informally worded as follows: “If Q does less than or the same as what P can do, then the assertion passes and Q is ‘safe’ relative to the specification P.” Alternatively, Q would be “unsafe” if it did more than P can do.

An example of a safety specification is the following assertion:

```
assert P [T= a -> STOP
```

Given P = a -> b -> STOP, the assertion says that the traces of ‘a -> STOP’ (i.e., {<>,<a>}) must be a subset of the traces of P in order for trace refinement to evaluate as true. The assertion is indeed true since the traces of P are {<>,<a>,<a,b>}. So P is trace refined by a -> STOP.

Failures refinement is a little more complicated. The following is a basic failures refinement assertion (or liveness specification) using the failures refinement operator, ‘[F=’:

---

[1] The expression “traces of a process” refers to the set containing all sequences of events the process can execute. For example, the traces of T = a -> SKIP [] b -> SKIP are {<>, <a>, <b>} since T can do nothing, ‘a’ alone, or ‘b’ alone.

```
assert P [F= Q
```

The assertion will be true if the “failures of Q” are a subset of the failures of P. A failure is defined as a pair (tr, X) where tr is a trace of a process P and X is the set of refusals for the process P that has already executed the trace tr. A refusal is simply the set of all events that would not synchronize immediately with the event offered by process P. For example,

```
P = a -> b -> STOP
```

would yield the following set of failures,

```
{ (<>,{}), (<>,{b}),
(<a>,{}), (<a>,{a}),
(<a,b>,{}), (<a,b>,{a}), (<a,b>,{b}), (<a,b>,{a,b}) }
```

Our liveness specifications could be informally worded as follows: “If Q is a description of all that P must do, the assertion passes.” More examples of both safety and liveness specifications with appear in section 4.3 of the ATM case study we will introduce later.

2.2 Terminology

In this section we attempt to clarify the definitions of the terms related to this thesis. Some of these terms have not yet been used in the thesis, but this will be a reference page as the terms surface. We begin with general terms in Table 1.

Table 1 General Terms

CSP	Communicating Sequential Processes—a formal process algebra for concurrent systems invented by Tony Hoare [Hoare, C. A. R. 1985].
csp12	A dialect of CSP invented by Cheng at the University of Victoria that was used as the original input notation for CSP++ [Cheng 1994].
CSPm	A dialect of CSP that is commonly used by CSP experts and is supported by commercial tools by Formal Systems Ltd. UK [Formal Systems].
FDR2	A refinement checker from [Formal Systems] used to check properties of specifications modeled in CSPm.
ProBE	An animator tool from [Formal Systems] used to interactively explore a CSPm model one event at a time.
Checker	CSP typechecker utility from [Formal Systems] that ensures datatypes are used correctly and increases confidence in the soundness of the syntax of the CSPm specification.
CSP++	Our tool for a synthesizable subset of CSPm that encompasses the cspt and OOAF to accomplish selective formalism for CSP and C++.
cspt	Translates CSP specifications into executable C++ code that targets the OOAF.
OOAF	Object-oriented application framework written in C++ that provides the execution semantics of CSP.

Since CSP++ was originally designed and documented for the csp12 dialect of CSP, there is some difference in terminology versus CSPm that can easily lead to confusion. It was decided, on the one hand, to leave the existing csp12-named classes, variables, and functions in the CSP++ source code to avoid breaking it, but, on the other hand, to talk about CSP specifications using current CSPm terminology. As a reference for future maintainers of CSP++, Table 2 below should suffice for explaining these differences.

Table 2 CSPm vs. csp12 Terms

CSPm Term	csp12 Term	Explanation
Set {..}	Set	These terms mean the same thing.
Closure Set {|..|}	-	In CSPm, we distinguish between a set and closure set. With the declaration channel c:{0..1} in CSPm, {|c|} is equivalent to {c.0,c.1}. Csp12 had no such thing. CSP++ supports both sets and closure sets with some restrictions (see section 3.5).
Process	Agent	A difference in terminology between CSPm and csp12. `Agent` is also a class name in CSP++ source code.
Event	Action	A difference in terminology between CSPm and csp12. `Action` is also a class name in CSP++ source code. For a more detailed description of an event see section 2.1.3 and section 2.1.4.
Event	Atomic/ Channel	Csp12 made a distinction between event types that is reflected in the CSP++ class names. Action is the parent class of Atomic and Channel in CSP++ source code. In CSP++, an Action using a '?' or a '!' is a Channel. Otherwise it is an Atomic. An Atomic can be with or without subscripts. The consequence of the distinction is that an Atomic cannot be mixed with a Channel in CSP++ as they can in CSPm. This restriction is given in Table 5 of section 3.5.
Compound Event	Datum	A compound event is an event with more than one part. A csp12 Datum was used to hold a number of values in conjunction with an Action (or event).
Values	Datum	CSP++ implements CSPm values with `Datum` objects.
Value	Lit	A csp12 Lit could hold a single value or a number of values (Datum). CSP++ continues to use Lit objects.

2.3 Related Work

The utilization of formal methods has encountered at least some resistance in software engineering groups. Formal methods have traditionally been an independent or isolated

part of the software development process—if they are a part at all [Sommerville 2000]. At least part of the challenge of utilizing formal methods directly in executable software development comes from the fact that formal specification languages are not full-fledged programming languages and programming languages are too semantically rich to verify directly. CSP++ does not stand alone as the only tool that attempts to meet the challenge of adopting formalism directly in the development of systems. There are many different approaches to incorporating formalism in both software and hardware synthesis. As these approaches are analyzed, some general categories of approaches emerge, but they often overlap.

If software engineers introduce elements of formalism into their software development process, they may do so in some of the following ways:

- Formal methods
- Special-purpose languages
- Libraries for general-purpose languages

Although various approaches to formal design do not always fit nicely into only one of these three categories, the categories provide us with a good way of classifying them and discussing their merits and drawbacks. We will now discuss these three ways of integrating formal elements into the software engineering process.

2.3.1 Formal Methods

Formal methods have not typically been designed with execution as their goal. Rather they are languages intended for reasoning about a system's properties and whose vocabulary, syntax, and semantics are formally defined and mathematically-based. Their

vocabulary, syntax, and semantics often vary greatly. Some examples of formal languages other than CSP include B [The B-Method], Z [The Z notation], and Promela [On-The-Fly, LTL Model Checking with SPIN].

Formal methods can be a starting point for synthesis. Just as FDR2 [Formal Systems] can be used to verify CSP specifications, so tools such as SPIN [On-The-Fly, LTL Model Checking with SPIN] can be used for verifying Promela specifications. After gaining confidence in the properties of a given specification, it can be automatically translated (like CSP to C++ via CSP++) to code for a general-purpose language (e.g., B to C using the B-toolkit [B-core (UK) Ltd]) or other desired forms. They can be translated into special-purpose languages (e.g. CSP to occam [Broenink, Jovanovic 2004]) that may be based on the formal method to begin with.

Sometimes, formal languages are translated to other formal languages. Perhaps one specification language is better suited to a certain problem domain. The tool csp2b [Butler 1999] is used to translate CSP to B since B weak in the area of modeling sequential activity (a strength of CSP) and can use CSP to model what, in B, would be more awkward. The Bandera [Corbett, Dwyer et al. 2000] tool goes the opposite way compared to CSP++ by extracting a formal model in Promela from general-purpose Java code so that verification can be performed on the already written source code.

2.3.2 Special-Purpose Languages

If someone would prefer to write an executable program directly instead of using a formal language but would still like to benefit from formalism, they may try a special-purpose language. When formal methods or general-purpose languages do not adequately

cater to the needs of specific problem domains, new special-purpose programming languages are designed, based on one or more formal methods, to meet those needs. Examples of such languages include the Esterel [Esterel: a Synchronous Reactive Programming Language], LOTOS [World-wide Environment for Learning LOTOS (WELL) - Introduction], occam, rebeca [Rebeca Home Page], and Rialto [Bjorklund, Lilius 2004].

Since these are programming languages, they have compilers and can be directly executed. These specialized languages can also be translated into other forms. Since they are formally based, they can often be translated with ease to a formal specification (e.g. rebeca to SMV or Promela [Sirjani, Shali et al. 2004]) that can be run through verification tools. Other verification tools can be run directly on the special-purpose language program with the necessary translation into formal models hidden from the tool user (e.g., Xeve [Welcome to the Esterel Verification Environment: Xeve] verifies Esterel). Finally, some toolkits are available that translate specialized programs into a general-purpose language (e.g. LOTOS to C using CAESAR toolkit [Fernandez, Garavel et al. 1992]).

2.3.3 Libraries for General-Purpose Languages

Software engineers may not want to bother with learning formal methods or special-purpose languages and would prefer to use libraries of functions or classes based on formal semantics to increase their confidence in the software they produce. Some research groups, like one at Kent, designed CSP-based libraries for Java (JCSP [Welch, Martin 2000]), C (CCSP [Moores 1999]), and C++ (C++CSP [Brown, Welch, Prof. Peter H. 2003]). Similarly, the University of Twente has developed three libraries for the same

languages called CTJ, CTC, and CTC++ [CSP for Java, Broenink, Jovanovic et al. 2002]. One open source C implementation of the CSP operational semantics called libcsp [Beton 2000, libcsp CSP on Posix Threads] is available on sourceforge.net, but does not appear to be under active development. Other Java libraries have been introduced including JACK [Freitas, Cavalcanti et al. 2002] in 2002 and Sea Cucumber[Jackson, Hutchings et al. 2003] in 2003. An older CCSP [Arrowsmith, McMillin 1994] also exists but there has been no known work on it for many years.

2.3.4 CSP-Based Synthesis Tools

CSP++ is a combination of both the first and third approach. It fits the first since the C++ code is derived directly from CSP specifications, and the third because it generates code that is targeted for an OOAF that acts somewhat like the libraries listed in section 2.3.3. There are many reasons why the CSP specification is translated into C++ for the OOAF rather than generating assembly code directly. Translating for the OOAF makes the CSP++ tools more portable, since the translator would not need to be changed when porting to other platforms. Furthermore, translating for a higher-level code generation target like the OOAF is not only easier but it also allows the generated code to more closely reflect the CSP input syntax, thus making the output C++ code very readable. Below we discuss some of the other CSP-based tools that are being used in the development of systems.

2.3.4.1 Software Synthesis from CSP

Raju [Raju, Rong et al. 2003] developed a translator based on Mathematica that, like CSP++, can translate CSPm specifications to an executable form by employing the CTJ,

JCSP, and CCSP libraries. A feature comparison of CSP++ with Raju's tool is can be seen in Table 3 with 'X' indicating a supported feature.

Table 3 Comparison of CSP++ and Raju for supported CSPm features

FDR2's CSPm Features	**Raju's CSP-to-**			**CSP++**
	CTJ	**JCSP**	**CCSP**	
Comments: --	X	X	X	X
Comments: {- ... -}		X	X	X
Integer data	X	X	X	X
Declarations	X	X	X	(1)
Process definitions	X	X	X	X
Recursive processes	X	X	X	X
Parameterized processes: P(2,i)				X
Prefix: ->	X	X	X	X
Channel I/O chan?data, chan!data	X	X	X	X
Channel I/O chan?d1.d2, chan!d1.d2	X	X	X	X
If ... then ... else ...	X	X	X	X
External choice (alternative): []	X	X	X	X
Interface (sharing) parallel: [|{|...|}|]	X	X	X	X
Interleaving parallel: P|||Q				X
Sequential composition: P;Q				X
Event renaming: [[e<-f]]				X
Event hiding: \{e}				X
Note (1): not needed for synthesis (treated as one-line comments)				
Not supported				
Boolean guard: &	Linked and alphabetized parallel			
Replicated operators: @	Interrupt: /\			
Untimed timeout: [>	Sequences and sets			

One key difference between the CSP libraries and the CSP++ OOAF is in the way external choice is handled. CSP++ implements a try-and-back-out mechanism where each

event in the choice can be tried until one of them completes execution—rolling back the unsuccessful choices. The libraries targeted by Raju's tool must commit to completing an event involved in a choice once it is tried and cannot rollback. CSP++'s handling of external choice is an important advantage over other tools for properly reproducing the semantics of CSP.

A hallmark of CSP++ is its ability to integrate user-coded functions (see section 4.4). The other libraries do not explain how to integrate code. Another tool has been developed, called gCSP [Broenink, Jovanovic 2004] (graphical CSP), that can generate CSPm, occam, or executable CTC++ code from a graphical representation of CSP.

The NASA Software Engineering Research Lab is currently working on a CSP-based solution to system design via a round-trip requirements-based formal development. The tool they call R2D2C ("Requirements to Design to Code") [Hinchey, Rash et al. 2005] would take system development from requirements to a provably equivalent mathematical model that can in turn be translated into software. Requirements are provided in a constrained natural language as scenarios from which traces of events are generated. CSP statements are inferred from these traces and can then be analyzed and automatically translated to code or instructions for a robot. R2D2C is in its preliminary stages and many of the tools still need to be developed. Since the CSP will not be written by humans and is inferred from traces, it will not generate very readable code and may be difficult to extend with user-coded functions like CSP++ does.

2.3.4.2 Hardware Synthesis from CSP

Future CSP++ research is intended to involve software/hardware codesign where software and hardware are synthesized from a CSPm specification. There already is a tool that translates CSPm to Handel-C [Phillips, Stiles 2004, Stepney 2003]—a hardware description language that can be used to program a Field-Programmable Gate Array (FPGA). The formal language B can also be translated to C as well as VHDL [Bjorklund, Lilius 2004] although it is not explained how or whether partitioning between software and hardware is done.

2.3.5 Application Areas for CSP-Based Systems

Apart from the small case studies we have done using CSP++ [Carter, Xu et al. 2005], NASA has several specific application areas for formal design using CSP including

- Sensor Networks: the ANTS mission will send out 1000 small sensor spacecrafts that make analyze asteroid composition—a complex task well-suited to modeling with CSP.
- Expert Systems: the NASA ground control center expert system for the POLAR spacecraft uses written rules that can be generated from CSP.
- Robotic Operations: Instructions for servicing the Hubble Space Telescope in space must be correct. Instructions could be generated from CSP.

In addition to what NASA is doing, the University of Kent have demonstrated their JCSP library with a simple steam boiler case study [McEwan 2004]. CSP is inherently suited to systems that are concurrent because of the way we can compose processes, communicate between processes, and execute process events sequentially.

Chapter 3

Reengineering CSP++ for CSPm

CSP++ was originally designed to work with the csp12 dialect of CSP. Case studies for CSP++ such as the DSS were written in csp12 and did not have any tools for verification, simulation, and syntax-checking such as those provided for the CSPm dialect by Formal Systems [Formal Systems]. In order to design case studies that could also be directly formally verified, we were left with two options: (1) either translate CSPm specifications to csp12 specifications, or (2) change the cspt [Gardner 2000] translator to accept the CSPm dialect.

The first option did not seem to be a good solution long-term since csp12 is unpopular and limited. CSPm is also more complex syntactically and semantically, with operators and abilities such as multilevel synchronization that could not be handled in terms of csp12 semantics. The second option, to reengineer CSP++ for CSPm, was not trivial but appeared possible, and necessary to make CSP++ useful and appealing for software engineers.

This chapter describes the reengineering of CSP++ for a synthesizable subset of CSPm in detail. We begin by discussing the theoretical issues that were resolved to make CSP++ syntactically and semantically faithful to CSPm. We will present the “best compromise” policies for CSP++ that were established. Following the theoretical treatment, the specific implementation decisions for the translator and framework will be explained. Finally, the restrictions and limitations of CSP++ will be summarized.

3.1 Theoretical Issues

As it was our aim to provide a useful, powerful, and fast tool, there were a number of trade-offs that were considered in reengineering CSP++. Generally, CSPm is more extensive in its support of features than the former version of CSP++. To determine which features to support, we filtered based on how useful the feature would be, whether or not the feature would introduce nondeterminism (not useful for synthesis), and whether or not it was worth the cost of implementing the feature. Those features that we deemed appropriate for CSP++ were chosen to be part of the synthesizable subset of CSPm to which CSP++ would conform. Other features, for reasons later described, were left out of this version of CSP++.

Of particular importance was that CSP++ accept specifications written for verification in FDR2 without hand massaging. This implied that features that FDR2 required for a verifiable specification be accepted even if they were not needed in CSP++. Furthermore, the specification used by CSP++ could have no extensions defined that would make it incompatible with FDR2. We have used the FDR2 manual [FDR2 User Manual] as our CSPm guide as it explains the intricacies of all the FDR2 features. We will now discuss the specific theoretical issues that needed to be addressed to establish “best compromise” policies. From the standpoint of CSPm conformance, we considered support for all variations on data types, events, process definitions, styles of composition, synchronization and communication semantics, and various other operators. These issues had implications on UCF integration. Finally, those CSPm features left unsupported are explained and justified.

3.1.1 Data Types Supported

CSPm has many predefined data types as well as the capability of defining custom data types. As we have seen in section 2.1.3, data type values, such as integers, can be used in compound events. They may also be used in conditional statements, parameterized processes, and other data type definitions. `datatype`, `nametype`, and `subtype` declarations are used to introduce custom types for CSPm specifications. Some data types can be quite simple while others are more complex. In section 2.1.6 we saw the use of `nametype` to refine the definition of `Date` with `Day` and `Month`. This could also have been written:

```
datatype Date = {0..31}.{0..12}
```

to indicate that a channel with type Date needs two integers in the ranges defined.

Enumerated data types can be defined in the following way:

```
datatype Number = zero | one | two
```

indicating that a channel of type Number can either have the value zero, one, or two.

The use of `subtype` definitions are not well explained in the FDR2 manual. Therefore, we allow them to appear in CSPm input, but ignore those definitions since they are not needed by CSP++ anyway.

Data types are further complicated by set, sequence, and tuple types. FDR2 supports sets that can not only be values for channels but can be passed as arguments to processes. The following example shows how sets can be used as data types:

```
channel pairUp: { {1,2},{2,3},{1,3} }
P = pairUp.{2,1} -> SKIP
```

The `channel` declaration enables the event ‘pairUp.{2,1}’ to be executed. Note that the value {2,1} is the same as {1,2} since they are set data types.

Sequences cannot be used in events but can be passed as parameters and have values extracted from them. Consider the following queue implemented using sequences as parameters:

```
channel dequeue, enqueue: {1..100}
QUEUE(s) = if s == <> then enqueue?x -> QUEUE(<x>)
           else enqueue?x -> QUEUE(s^<x>)
         [] dequeue!head(s) -> QUEUE(tail(s))
```

In this example, a sequence representing a queue’s current contents is passed as parameter s to the process QUEUE(s). If there are no items in the sequence—that is, the parameter s == <>, the empty sequence—the specification only permits an ‘enqueue?x’ event, which leaves the queue containing <x>. Otherwise, the specification also allows a choice of enqueue or dequeue operation. Whichever event is executed, the new sequence is passed recursively to QUEUE(s) to await the next event for the Queue. The FDR2 manual explains the use of various operators for sets (element-of, union, intersection, subtraction, etc.) and sequences (concat, etc.).

Tuples are yet another data type, formed by a comma-separated list of values in parentheses, such as (3,20). Combining set notation with tuples can produce a cross-product effect. For example, the following declaration:

```
nametype T = ({0..2},{1,3})
```

has the same meaning as `T = {(0,1),(0,3),(1,1),(1,3),(2,1),(2,3)}`, thus defining members of T as any of 6 tuples.

These examples demonstrate the abilities of CSPm to specify various data structures. However, implementing CSP++ support for all of these would require significantly more research and will have to wait for later versions of the tool.

In CSP++, we support integers alone since they have shown themselves to be useful and sufficient in case studies so far. For example, currency values would at first glance appear to call for floating point numbers, but a value of $34.96 could be represented as a pair of integers or simply as a single integer 3496.

Support for additional data types is not necessarily difficult to add in the future because of the object-oriented architecture of the CSP++ framework. At this stage, it is unnecessary for the translator to recognize `datatype`, `nametype`, and `subtype` statements in order to carry out synthesis, therefore we ignore these declarations for now.

We experimented with implementing a mechanism for supporting simple enumerated `datatype` declarations like Number above in a way that we describe later. However, the problem with only supporting simple enumerated data types and not supporting others is that the `datatype` keyword is no longer ignored and non-conforming `datatype` declarations must also at least be recognized, which is quite complex. It was considered unacceptable to give the specification writer an error message saying that the data type is not in the correct format for CSP++ when it is correct for FDR2. We do not want to inhibit specification writers from declaring for verification purposes data types such as "`datatype Set = {0..3)`" that, when ignored by CSP++, cause no problems in synthesizing the non-datatype statements.

3.1.2 Events Supported

As we know from section 2.1.3, CSPm events can be simple or compound. An event begins with a channel name optionally followed by any number of dot-delimited values. In CSPm, all events must be declared using `channel` declarations as seen in section 2.1.6. When we declare `channels` we give the channel name, optionally followed by a colon and a number of dot-delimited data types.

CSPm and csp12 differ in their interpretation of events. The confusion stems from the original explanation of communication where, in his chapter on communication, Tony Hoare explains that the functionality of communication operators can be achieved without using the extra notation of '?' and '!' and that channel I/O is essentially a meta-concept layered on top of events introduced for more powerful reasoning and convenience for certain applications [Hoare, C. A. R. 1985]. In CSPm, events are a full record of a channel name, subscripts (if any), and communicated data values (if any). In CSP++, events (Action objects) are just a channel name and subscripts (if any), with communicated data handled separately. The result in CSP++ is that the Action class has two subclasses—those that are intended for communication (Channel objects) and those that are not (Atomic objects).

Several policies on events that work well for software synthesis are listed below along with reasons why each restriction was established:

- Continue to distinguish between Channel and Atomic Actions. An Action cannot be both an Atomic and a Channel. For example, 'c.1' is different from 'c!1' in CSP++, even though it represents the same event in CSPm.

This policy was established since Channel and Atomic Actions each have distinct purposes in synthesis and blending them makes it difficult to distinguish between what event is doing I/O and what is not.

- Allow Atomics and Channels to have subscripts.

Csp12 only supported subscripts in Atomic events. Since CSPm made no such distinction, it only seemed consistent to support Channel subscripts, too.

- Restrict channel I/O to using exactly one communication field operator '?' or '!' (i.e., no mixed mode communication).

CSPm allows "mixed mode" communication like 'c?x!2' synchronizing with 'c!1?y'; this is dealt with further in section 3.1.5. We do not allow mixed mode communication because it significantly increases the complexity of communication, and traditional one-way I/O is sufficient for applications.

- We enforce that events appear with all subscripts and data items explicitly listed (and thus matching in number), except in sets of events (see later in this section).

In CSPm, it is possible to have a channel like 'c?x' synchronize with 'c!1.2', binding 'x' to the value '1.2'. However, having a variable hold more than one value makes the specification more difficult to understand.

Specifically, these decisions lead to the following syntax conventions for CSP++ Actions:

- Atomic Actions are described by chan[.s]*, where 'chan' is a channel name and 's' (subscript) is any integer or bound variable. []* indicates that what is within is

optional and can be repeated. In CSP++, the number of subscripts is presently limited to 10, as are the number of data values per event.

- Channel output is described by chan[.s]*!d[.d]*, where ‘chan’ and ‘s’ are as above and ‘d’ (data) is any integer or bound variable.
- Channel input is described by chan[.s]*?v[.v]*, where ‘chan’ and ‘s’ are as above and ‘v’ (variable) is any free variable.

These conventions are fully compatible with CSPm, and at the same time force the designer to clearly distinguish subscripts and data values, which in CSPm can be quite ambiguous.

Sets of Events

Sets of events are needed in specifications in conjunction with the composition and hiding operators. The following process illustrates sets of events in the context of both composition and hiding, `P [|{|c,d|}|] Q \{|d|}`. In this process, P and Q synchronize on any event starting with ‘c’ or ‘d’ because of the `{|c,d|}` event set. All synchronizations on events starting with ‘d’ are hidden by the ‘\’ operator due to the `{|d|}` event set. “Hiding” means that the occurrence of the event does not propagate outside the context of the expression, and does not appear in the process’s trace. One use of hiding an event is to enable other processes outside the scope of the hiding to reuse the event name without synchronizing with the hidden event, much like making a variable name local to a function. This technique facilitates the modular reuse of process definitions by other specifications.

The csp12 version of CSP++ also used event sets for renaming, but in CSPm, this operator takes pairs of channel names: `[[a<-b]]`, pronounced “a becomes b” which has the effect of renaming any channels with the name ‘a’ to ‘b’. So, for example, ‘a.1’ would become ‘b.1’.

Thus, event sets are used by CSP++ at translation time to generate code. These sets should not be confused with using sets as run-time data values, which CSP++ does not presently support.

Writing out event sets exhaustively can be burdensome (or impossible, for `channels` with infinite data types), therefore CSPm has a “set closure” notation ‘{|..|}’ to provide the effect of listing all the productions of a channel name in the set. Since CSP++ did not have this feature, the reengineered CSP++ for CSPm needed a way to distinguishing between {..} and {|..|} sets. In CSP++, we distinguish between the two notations so that their operational semantics are the same as CSPm’s with the following restriction:

- Allow only bare channel names in sets (i.e., disallow any subscripts or data values).

We enforce this restriction because of the overhead necessary to enable CSP++ Channels, designed to synchronize on the channel name and subscripts only, to match data values as well. The only reason that a specification writer would want to explicitly write out subscripted events in a set would be because they would want to synchronize on, hide, or rename events with the same channel name but different subscripts. We felt that our restriction was reasonable for a number of reasons:

- In all the examples we have encountered so far, none have required subscripts in sets.
- The same effect can be achieved by another way by, say, replacing a.1 with a_1 and using a_1 in the set.
- Channels only synchronize if there is a closure set (i.e. unless all the individual expansions of the channel are explicitly listed in the set) or if the event value gets bound somewhere else in the tree and the explicitly listed synchronization set contains the event with that bound value. It is not likely that the specification writer would want to explicitly write every event value in the set for large sets.

It is possible that we could remove this restriction in the future if an efficient way to allow subscripts or data in sets (or lists for renaming) is found.

3.1.3 Processes Supported

Section 2.1.1 introduced the idea of processes. The termination processes, SKIP and STOP are supported in CSP++. Parameterized processes are also supported, but not to the full extent that they are in CSPm. CSPm allows data types other than integers to be passed as parameters while CSP++ restricts parameters to integers. Since we only support integers at this time, this is not an unreasonable restriction. In the future, if support for other data types is introduced, the implications of allowing them in parameterized processes should be considered.

CSP++ supported csp12's "fixed point" expression. Defining a process using the csp12 keyword 'fix' would essentially provide an inline process definition. CSPm lambda terms, which are nameless functions, may make some use of the fixed point expression in

the future. For this reason the code for ‘fix’ remains in the source files, but is deactivated by commenting out.

3.1.4 Composition

Processes can be composed in three ways in CSP++, as mentioned in section 2.1.2. Sequential composition in CSPm and csp12 are identical, using the ‘;’ to separate processes that are to be executed sequentially. Independent composition ‘|||’ is also unchanged as it can be written as it was in csp12 CSP++ with up to 8 processes composed together.

Synchronized composition is handled differently in csp12 and CSPm. The difference is subtle and important, and proved to be the single most difficult problem to solve. In csp12 CSP++, there was a mechanism for multiple processes synchronizing together on the same events. If processes P, Q, and R were to synchronize on event ‘c’, it would be written in csp12 as (P || Q || R)^{c}. Synchronized composition in CSPm is expressed in terms of only two processes, since ‘||’ is a binary operator. Although this is the case, a CSPm event can be used for synchronization between multiple parties in multiple levels of hierarchy as in (P[|{c}|]Q) [|{c}|]R, whereas csp12 CSP++ could not do this. The point is that all instances of ‘c’ must be able to synchronize whenever they appear in the CSPm process hierarchy, whereas in csp12, the possibility of synchronization was restricted to processes at the same level. This restriction in csp12 severely limited the complexity of the case studies that could be developed, and, more importantly, was in direct contradiction to FDR2 semantics. This led to CSPm specifications that verified correctly behaving differently when synthesized by CSP++.

Furthermore, multiparty synchronization in csp12 CSP++ could not handle channel I/O, because of the assumption that channels were strictly for point-to-point use between two processes. The effect of multiparty synchronization in CSPm is that all the parties synchronize on the same event and perform it together once. If the synchronization involves channel I/O, data will communicated during synchronization. The intricacies of channel I/O are discussed in the next section.

CSPm's pattern matching strategy for synchronization is very flexible but would require significant overhead to implement in CSP++ and is not necessary for software synthesis. The following restriction was established for synchronization.

- All subscripts must match in order for a synchronization to occur (i.e., everything before the communication field operator, if applicable, must match). For example, the translator would reject 'c.1.2' and 'c.1!2' in the same specification even though these could synchronize in CSPm.

3.1.5 Communication

CSPm uses a full pattern matching semantics for communication rather than what would be understood as traditional channel I/O. As outlined in section 2.1.4, Hoare adopted the convention that communication be point-to-point, unidirectional, synchronous, and non-buffered. The csp12 version of CSP++ was implemented with this understanding except that communication was permitted to be bidirectional, although not in the same event (i.e., half duplex). CSPm permits bidirectional communication even in the same event (i.e., full duplex or mixed mode communication) as in this skeletal example (which does not work exactly as shown), c?x!2 || c!1?y. Figure 1 illustrates the spectrum of

communication conventions from CSPm's interpretation on the left to Hoare's original definition on the right.

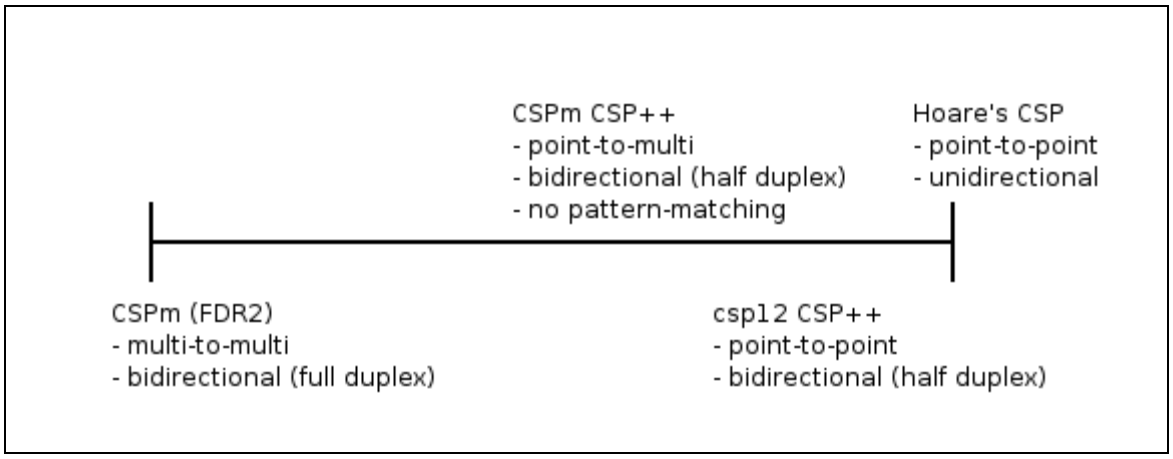


Figure 1 Support for CSPm and Hoare's CSP.

In the new CSP++, we adopt a middle-of-the-road convention for event communication that adopts a subset of the CSPm communication functionality. We decided to restrict our CSPm version of CSP++ in the following way:

- We allow broadcasts from a single output to multiple inputs rather than the many-to-many approach of FDR2's interpretation of CSPm. More than one output in the same synchronization is illegal in CSP++ (e.g., 'c!1' cannot synchronize with another'c!1', or 'c!2', or anything but 'c?variablename').

While one-to-many broadcast communication can be useful in software systems, many-to-many "broadcasts" are non-intuitive in their intent, and tend to obfuscate specifications.

CSP++ provides default behaviours for I/O that is not synchronized within a specification, such as output with no corresponding input, or multiple inputs with no output. The default behaviours interact with the console:

- An output operator without any corresponding synchronized input operator outputs its values to **stdout**.
- Similarly, an input operator without any corresponding synchronized output operator inputs from **stdin**—provided that the input operation is not hidden. Hiding input without providing synchronized output makes the CSPm specification nondeterministic.
- A number of input operators can synchronize together without output. In this case, the resulting CSP++ behaviour is that there is one read from **stdin** and that value is distributed to all the inputs.

Channel Data

CSPm and csp12 both allow for communicating multiple data values over a single channel, but their syntax is very different. In the csp12 Disk Server Subsystem (DSS) case study, a client would make a request to the disk server with the client number and the block number using the output operation ‘ds!req(1,100)’ and the server would receive the client’s request with the input operation ‘ds?req(_cl, _blk)’. I/O operations that transferred multiple data values, such as ‘req(...)’, are what csp12 called “datums.” Instead of using datums for communication, CSPm uses any number of variables or values strung together by dots following the channel name. CSPm would accomplish the

above disk request with operations like ‘ds!1.100’ for the request and ‘ds?cl.blk’ to receive it. Since CSPm does not provide for datums, CSP++ will not support them either.

However, CSPm’s way of handling multiple data values is not simple. CSPm allows a single variable to hold multiple values. Consider the following CSPm specification:

```
channel c,d: {0..1}.{0..1}
P = c!0.1 -> d?x.y -> SKIP
Q = c?z -> d!z -> SKIP
SYS = P [|{|c,d|}|] Q
```

Process P begins by outputting 0 and 1 on channel ‘c’. Process Q synchronizes on the same channel ‘c’ and stores ‘0.1’ in the variable ‘z’. When process Q outputs ‘z’ along channel ‘d’, synchronization takes place with process P’s channel ‘d’—transferring the two values in ‘z’ to the variables ‘x’ and ‘y’, respectively. At first it may appear that processes can be defined that accept any number of values and store them in a single variable. However, the reason we are able to do this in the first place is because the channels are clearly declared ahead of time. In our example above, we know that ‘c’ and ‘d’ must hold 2 values from their declaration.

CSP++ does not process declarations nor fully enforce the consistent use of channels. Below we see how channel ‘c’ was used in csp12 CSP++ to hold a datum with 2 values and to hold a single value.

```
P ::= c!d(0,1) -> c!1 -> SKIP.
Q ::= c?_z -> Q.
SYS ::= (P || Q)^{c}.
```

The first time Q synchronizes on channel c, variable _z will receive the datum d(0,1). The second time, _z will receive just 1. The reason why this worked in CSP++ is that if we input a single variable in CSP, it is translated as a FreeVar object. This container object allows any Lit to be input (including a Datum). However, if we removed process P and

left Q to interact with the environment, CSP++ would not know whether the variable '_z' (as a FreeVar) was expecting a single value or more. In this case CSP++ would only allow a single variable to be entered via **stdin**. So, datums had inconsistencies in csp12 CSP++.

To illustrate another difference between what CSPm and CSP++ could do, consider the following CSPm statements:

```
channel c:{1..3}.{1..3}.{1..3}
P = c!1.2.3 -> SKIP
Q = c?x.y -> SKIP
SYS = P [|{|c|}|] Q
```

Process Q's channel 'c' would input '1.2.3', but 'x' would get the 1 and 'y' would get '2.3'. CSPm can do this because it knows how many values channel 'c' needs to be complete. This is an example of 3 values mapped to 2. CSP++ could have mapped *n* values to 1 but not *n* to *m* like we see in the example above. So, again we see that allowing communication between channels with different numbers of values is only unambiguous given a formal definition of the channel.

Until CSP++ processes declarations, if ever, the sure way to keep CSP++ as a subset of CSPm would be to enforce the following rule.

- A variable can hold only one value (e.g., 'c!1.2' cannot synchronize with 'c?x' since 'x' cannot hold '1.2' but 'c?x.y' can).

Enforcing this rule ensures that there are no "hidden" values (i.e., multiple values in one variable) and makes it clear where and how values are stored. The only benefit of not having to write match variables to values one-to-one is convenience.

3.1.6 Operators Supported

Beyond the composition and I/O operators described above, CSP++ already supports the following operators of CSPm:

- prefix ‘->’
- conditional ‘if *cond* then ... else ...’
- hiding ‘\’
- renaming ‘[[...<-...]]’

For some operators, the csp12 syntax was slightly different from CSPm, but easy to change to the latter.

In addition to the operators above, CSP++ supports the CSPm relational operators (i.e. ‘>’, ‘<’, ‘>=’, ‘<=’, ‘!=’, and ‘==’) that can be used with the conditional operators described above. The usual arithmetic operators were already supported except for the addition of the ‘%’ modulus operator.

As for a choice operator, the csp12 version of CSP++ used '|' whose closest relative in CSPm is the ‘[]’ external choice operator. There is a subtle difference between the '|' choice operator in csp12 and the '[]' external choice operator in CSPm in that '[]' can operate on process names (e.g., P [] Q) while ‘|’ cannot. This is because ‘|’ depends on the first event of each alternative to be exposed in prefix form (a -> P). Therefore, in CSP++ we only allow processes of the a -> P prefix form to be involved in choice. However, we also allow processes with conditional operators to be involved in choice provided that the resulting processes are of the prefix form.

The problem with implementing P[]Q in CSP++ is that it can be hard to tell at translation time what the first event of each alternative will be. Detecting those first events at runtime is very complicated and may require significant additional overhead. In the future, a possible way around this problem is to algebraically manipulate the process definitions into their "head normal form" (see Roscoe), which exposes their first events.

Although we do not implement inherently nondeterministic operators such as internal choice '|~|' and untimed timeout '[>', other operators may appear in nondeterministic forms which may or may not be detected by CSP++. For example, the process a->P [] a->Q is nondeterministic but is not detected as such by CSP++. In this case, code would be synthesized, but P would always result when 'a' occurs. It is left to the specification writer to verify the specification for nondeterminism before using the CSP++ tool.

We restrict some forms of external choice with hiding and renaming that are not restricted in CSPm since they are not useful and they are complicated to implement in CSP++. When the initial events of the alternatives are hidden events, external choice is, in effect, changed to internal choice. For example, (a -> P)\{a} [] b -> Q is rejected at translation time since hiding prevents the initial event 'a' from being exposed. Although (a -> P [] b -> Q)\{a} is permitted, P could never be chosen since 'a' is not offered externally. The following renaming case, (a -> P)[[a<-aa]] [] b -> Q, is not permitted event though it is clear that 'aa' is the event to be exposed. While such operations can syntactically be coded, they represent degenerate cases that would be of no use in practical specifications. We do allow renaming in the following form, (a -> P [] b -> Q)[[a<-aa]].

3.1.7 Other Supported Features

CSPm has a notation for supporting block comments ‘{- ... -}’as well as end-of-line comments ‘--’. CSP++ supported csp12-style end-of-line comments using the ‘%’ operator but now uses both of CSPm’s comment notations. Note that block comments cannot be nested in CSPm.

To allow specification writers to use the same specification for verification and synthesis, we simply ignore those things that are not needed for synthesis. The `channel`, `datatype`, `nametype`, and `subtype` declarations and `assert` statements in a specification are important for FDR2 verification but can be ignored by CSP++. They are ignored provided that a single declaration is not broken up across multiple lines.

Another declarative construct is used in conjunction with the input ‘?’ operator in CSPm to limit the type and range of data values that will be accepted. This is called constrained input. Consider the following specification.

```
channel c: {0..3}
P = c?x : {1..2} -> SKIP
```

The channel ‘c’ is constrained to allow only input from values between 1 and 2 rather than between 0 and 3. However, for synthesis purposes, this can also be ignored. The consequence is that we cannot control the type of data that is input. This is not a problem for the CSPm specification itself but raises some issues for User-Coded Function Integration, which will be discussed next.

3.1.8 UCF Integration

Since we ignore declarative statements, we rely on the verification tools to ensure that the specification uses valid ranges of data. However, once UCFs are introduced, any data that

is input via a UCF has the potential of falling outside of the declared ranges. Moreover, if the specification is left to input from stdin (default behaviour), we may input invalid data. If either of these inputs obtain invalid data, that data propagates into the formal backbone and could cause undesired behaviour. The result would be that we could no longer assume that our verified properties continue to hold. There are basically two ways to deal with the problem of input validation:

1) Allow the specification to input a value of any range and then check the value in the specification, rejecting illegal input.
2) Write UCFs that check the input before returning the value back to the backbone.

More will be said about these options, including some recommendations, in section 4.4.

3.1.9 Unsupported Features

Besides the restrictions and conventions presented in the preceding sections, there are other features from CSPm that were not included in this version of CSP++ for CSPm syntax. We have classified them into three categories.

(1) Some features would be useful to have and can be implemented *later* as future work.

(2) Other features may be useful but the cost versus benefit ratio is so low that they are *not worth implementing*.

(3) Still other features are *not useful for synthesis*, including those that introduce nondeterminism, and will not be supported.

(1) Useful for later

Replicated operators are available in CSPm to simplify the writing of many external choices or compositions strung together. For example, this replicated interleave operator, ||| i:{0..2} @ P(i), would have the same effect as P(0) ||| P(1) ||| P(2). We do not yet have a mechanism for doing replication but it may prove to be useful later on.

We showed some example of sets, sequences, and tuples in section 3.1.1. These data types may be useful for synthesis.

The boolean guard operator '&' acts like the conditional operator. 'count >= 10 & P' would be equivalent to 'if count >= 10 then P else STOP'. CSPm's interrupt operator '/\' would be very useful so that processes like P/\Q would allow the execution of P to be interrupted at an arbitrary point and have control transferred to Q when the first external event of Q is offered.

(2) *Not worth implementing*

CSPm supports the alphabetized parallel operator 'P [A||B] Q' described in section 2.1.2 but CSP++ does not. The operator is intended to be used with the alphabets (i.e., all the events a process does) of the two processes in synchronization. These alphabets are not calculated from the process definitions by FDR2 but must be explicitly written out by hand in the specification. The synchronization occurs based on the intersection of the two sets, which, in effect, creates a set interface. The interface parallel operator uses an interface to begin with. For software engineers, interface is much more understandable because the events of interest are listed plainly. The alphabetized parallel operator could be supported by calculating the set intersections at translation time. There is no clear

advantage to using the alphabetized parallel operator over the interface parallel operator, so it is not worth implementing at the moment.

Although full pattern matching would create great flexibility in CSPm specifications, the run-time overhead involved makes it not worth the effort. Furthermore, CSP++ already provides the kind of channel I/O needed for system building.

(3) Not useful for synthesis

Any operators that are inherently nondeterministic are not be implemented. Nondeterministic operators are used to model processes beyond the control of a system. They may be useful for environmental model processes (see section 4.1) but these processes are removed for synthesis anyway. The prominent example if CSPm's internal choice operator '|~|' where one of the alternatives is arbitrarily chosen outside the control of the process. The so-called "untimed timeout" operator '[>' is also nondeterministic because it is defined in terms of the internal choice operator. It allows a process to offer an event for an undetermined amount of time before taking away the offering and continuing with another process.

3.2 Translator Changes

The work of reengineering the translator was carried out in two phases. This yielded a version of CSP++ that could be used for CSPm case studies and training as soon as possible, while deferring the most difficult problems until they could be properly addressed.

The first phase, V4.0, primarily involved syntactic changes to accommodate CSPm. It was, in effect, csp12 disguised with CSPm syntax. There were still some features in CSPm that were desirable for synthesis, but they were left for the second phase (V4.1).

V4.0 met our initial goals and enabled us to create case studies with much greater ease. As case studies were created with different combinations of features, many bugs were fixed in CSP++ and the tool became increasingly robust.

As V4.0 still handled CSPm with csp12 semantics, unpredictable results would occasionally surface. Limitations, such as V4.0's inability to handle multilevel synchronization, were unsatisfying when developing more complex case studies. These discrepancies were fixed in the second phase of development.

The following section describes the various ways that were considered for reengineering cspt. The rest of this section looks at the specific changes that were needed in the translator to agree with the "best compromise" policies described in section 3.1.

3.2.1 Reengineering the Front-End of CSP++

The cspt translator was originally built using the flex lexical analyzer and the bison parser to build an object-oriented parse tree used for code generation. Modifying the cspt translator's flex and bison files was one way to reengineer CSP++ for CSPm, but other options were considered as well.

Special arrangements with Formal System made their flex and bison files for FDR2 available. These had the advantage of being fully compatible with CSPm. It was worth considering the possibility that these files could be used to construct an FDR2-style of OO parse tree or changed to construct cspt's parse tree.

In addition, FDR2 contains a Tcl program named fdr2tix that parses a CSPm specification, builds some internal representation of the specification, and can be queried to obtain information about the specification. The option was considered of offloading the parsing and structure-building to fdr2tix, so that cspt need only extract the necessary information from fdr2tix to generate C++ code for CSP++.

In summary, reengineering the front end of CSP++ could have been attempted in at least three different ways:

(1) Use FDR2's parser

(2) Use FDR2's front-end processor, fdr2tix

(3) Modify cspt

These approaches are considered in turn below.

(1) Use FDR's parser

Although using the same parser as the verification tool intended to be supported was appealing, there turned out to be significant roadblocks. The parser was little documented and reverse engineering proved difficult. Also, FDR2's OO parse tree was so different from cspt's that much of the translator's code would have needed to be thrown away or changed significantly. Furthermore, FDR2 supports more CSPm constructs than what we consider useful for synthesis.

(2) Use fdr2tix

Fdr2tix is a Tcl program that provides access to the underlying object model of FDR2. Incorporating fdr2tix into cspt was considered in order to harness the existing processing power of fdr2tix rather than have cspt replicate it.

Fdr2tix allowed the extraction of state machine objects (ISMs) by compiling processes. Compiling a process would yield one or more ISM(s) by depending whether or not the process was a combination of other processes. State machine representation from each ISM could be obtained by using the "ism transitions" command. These state machine objects provide information concerning how a particular event (identified by number) causes the machine to move to a new state. The names of these events could be found by using the "ism event *eventnumber*" command.

The fdr2tix ISMs were useful for learning about the process tree and observing the events that occur. However, we faced at least three obstacles:

(a) State machine representations did not clearly reflect the original CSPm textual specification since fdr2tix reduced process definitions to equivalent state machine representations. As it was our desire to have CSPm specs translated to readable C++ code, this was unsatisfactory.

(b) Reduced ISMs made it impossible to determine if choice or other conditional operators had been used, and these had to be translated into run-time operators.

(c) The idea of “input and output” was abstracted as each occurrence of an event appeared only as compound events separated by dots. It could not be determined which events were I/O operations and which were not.

These reasons made it clear that fdr2tix was not a good candidate for integration with the cspt translator.

(3) Modify cspt

Our own CSP++ parser for csp12 worked well and was well understood. Although there was a gap between csp12 and CSPm both syntactically and semantically, it appeared that it would not be too difficult to modify the csp12 syntax tokens to match the CSPm syntax.

We also considered combining two or more approaches. One way was to have an initial trivial parse phase that would process comments aimed at synthesis control and extract CSPm source code for readable generated code. The second phase would let fdr2tix process the specification so that we could extract information about the processes, events and other structures, outputting the resulting CSP++ code. However, the limitations of fdr2tix (mentioned above) would not permit this combined approach either.

Therefore, as a result of the investigation of various options for reengineering the translator, it was decided that cspt must be modified and the other options be rejected. After a summary of the CSPm syntax supported by CSP++, the following sections will highlight the changes in way the translator now handles simple features, channel I/O, and data types.

3.2.2 Overview of cspt Translator

As the translator parses CSPm, BNF-like bison rules either create new ParseNode objects or add a token to an operand list in preparation for the creation of a new ParseNode object. These objects will be one of the ParseNode subclasses: PNcop (complex operators), PNtok (simple tokens), or PNcid (complex identifiers). ParseNode objects are built up hierarchically to form an object-oriented parse tree.

ParseNode objects may need to be prepared for later code generation. Preparation is done with the prep() virtual function and generation with gen(). If these functions must have special behaviours, they can be overridden for the specific needs of the particular ParseNode. Many ParseNode subclass constructors require access to symbol tables for storing and looking up names, and those tables are provided in the Symbol.h/.cc files.

The translation design is discussed in detail in Appendix D of [Gardner 2000]. Table 4 shows the BNF syntax for the CSPm now supported by CSP++, as well as the updated operation of the corresponding ParseNodes' prep() and gen() functions. Partial datatype support was added to CSP++, but is currently disabled until it can be fully supported in later versions. Csp12's old FIX operator is disabled but may be used for CSPm lambda terms [FDR2 User Manual] in the future.

Table 4 BNF syntax with corresponding parse node classes

Accepted CSPm syntax in BNF	Parent			Subclass Name	Pseudocode[1]		
	PNtok	PNcop	PNcid		ctor[2]	prep()	gen() and details for entries marked ">" (ctor = constructor)
	ParseNode				>	OK	ctor: store line number gen(): OK
	*	PNtok			{ }	-	-
		*	PNcop		{ }	Apply prep/gen to each operand in turn; stop on bad status	
			*	PNcid	{ }	>	prep(): apply to each arg/subscript; stop on bad status gen(): output name
<definition> ::= <signature> '=' <agent>		*		PNdefn	{ }	NC	prep signature and agent; use agent's symbol entry to gen AGENTPROC, arg #defines, and FreeVars (genAgentProc); gen agent body; "ENDAGENT" if needed; gen arg #undefs (genEndAgent)
<agent> ::= ('(' <agent> ')' | <prefix>	*see <prefix> below*						
| <agent> '[]' <agent> { '[]' <agent> }		*		PNchoice	>	-	ctor: continue only if all agents are prefix "Agent::startDChoice(n)"; set flag for PNinput (DatumVar gen); genPre actions; "Agent::whichDChoice()"; genPost agents

[1] Abbreviations: { } = no-op; - = default to parent's method; OK = no-op, return good status (0); NC = method is not called; "foo" = output "foo"
[2] Constructor: The obvious action of storing arguments in data members is not explicitly written out.

Table 4 BNF syntax with corresponding parse node classes

Accepted CSPm syntax in BNF	Parent			Subclass Name	Pseudocode[1]		
	PNtok	PNcop	PNcid		ctor[2]	prep()	gen() and details for entries marked ">" (ctor = constructor)
| FIX <UID> '.' <agent>[3]		*		PNfix	{ }	>	prep(): use agent's symbol entry to extract agent as subagent (makeSubAgent); change <UID> refs in subagent to new PNconstSub (changeConstRefs); gen subagent gen(): gen the PNconstSub
| <agent> ';' <agent> { ';' <agent> }		*		PNseq	{ }	-	gen each agent, flagging last one
| <agent> '|||' <agent> { '|||' <agent>		*		PNcompose	{ }	>	prep(): prep simple agents; complex: use agent's symbol entry to extract subagents (makeSubAgent), then gen gen(): "Agent::compose(n)"; "START" each agent; "WAIT" each agent
| <agent> '[|' '{|' <ID> {',' <ID> } '|}' '|]' <agent> | <agent> '[|' '{' <ID> {',' <ID> } '}' '|]' <agent> | <agent> '\' '{|' <ID> {',' <ID> } '|}' | <agent> '\' '{' <ID> {',' <ID> } '}' | <agent> '[[' <rename> { '.' <rename> } ']]'		*		PNenv	>	-	pre-ctor: if synchronization, new PNcompose gen the ActionRefs; "."; "sync()", "hide()", or gen PNrename; gen the associated agent; "Agent::popEnv(n)" if needed
| STOP	*			PNstop	{ }	-	"Agent::stop()"
| SKIP	*			PNskip	{ }	-	Set flag to get ENDAGENT generated

[3] FIX remains in the code base from csp12 CSP++ but is currently disabled in CSP++. It may reappear in CSP++ later if a corresponding CSPm construct is found.

Table 4 BNF syntax with corresponding parse node classes

Accepted CSPm syntax in BNF	Parent			Subclass Name	Pseudocode[1]		
	PNtok	PNcop	PNcid		ctor[2]	prep()	gen() and details for entries marked ">" (ctor = constructor)
| <ID> ['(' <exp> { '.' <exp> } ')']			*	PNconst	{ }	>	prep(): find in agentTable, get agentproc name via bindSig(args) gen(): "CHAIN", "START", or "START/WAIT" depending on context
| IF <exp> THEN <agent>		*		PNifthen	{ }	>	prep(): prep agent gen(): "if ("; gen exp; ") {"; gen agent; "}"
| IF <exp> THEN <agent> ELSE <agent>)		*		PNor	{ }	-	new PNifthen gen PNifthen; "else {"; gen 2nd agent; "}"
<prefix> ::= <action> '->' <agent>		*		PNprefix	{ }	-	gen(): - genPre(): gen action genPost(): gen agent
<signature> ::= <ID> ['(' <numvar> { ',' <numvar> } ')']			*	PNsig	>	>	ctor: find in agentTable, or insert new variant prep(): find signature in agentTable, set its symbol entry as the translation context; setup symbol entry to handle symbols for variant (prep) gen(): NC
<numvar> ::= (<NUM>	*			PNnum	{ }	-	output value

Table 4 BNF syntax with corresponding parse node classes

Accepted CSPm syntax in BNF	Parent			Subclass Name	Pseudocode[1]		
	PNtok	PNcop	PNcid		ctor[2]	prep()	gen() and details for entries marked ">" (ctor = constructor)
| <ID>)	*			PNvar	{ }	>	prep(): report to agent's symbol entry (addvar) with "global" flag if in subagent gen(): output var name, maybe globalized, obtained from agent's symbol entry (ref)
<action> ::= (<ID> ['.' <exp> { '.' <exp> }]			*	PNatomic	>	OK	ctor: find/insert in actionTable gen(): output name, gen subscripts
| <ID> ['.' <exp> { '.' <exp> }] '?' (<ID> | <ID> {'.' <ID> })		*		PNinput	>	-	pre-ctor: if > 1 <ID> after '?' then new PNdatumvar, otherwise new PNvar ctor: new PNchannel gen(): if datumvar, "DatumVar" temp "=" gen datumvar; gen PNchannel; ">>"; gen PNvar or temp
| <ID> ['.' <exp> { '.' <exp> }] '!' (<exp> | <exp> { '.' <exp> })		*		PNoutput	>	OK	pre-ctor: if > 1 <exp> after '!' then new PNdatum ctor: new PNchannel gen(): gen PNchannel; "<< ("; gen exp; ")"
			*	PNchannel	>	-	ctor: find/insert in actionTable, set actionType to AT_CHANNEL if necessary gen(): output name, gen subscripts
			*	PNdatumvar	>	-	ctor: find/insert in datumTable gen(): output name, gen subscripts

Table 4 BNF syntax with corresponding parse node classes

Accepted CSPm syntax in BNF	Parent			Subclass Name	Pseudocode[1]		
	PNtok	PNcop	PNcid		ctor[2]	prep()	gen() and details for entries marked ">" (ctor = constructor)
			*	PNdatum	>	OK	ctor: find/insert in datumTable gen(): output name, gen subscripts
<rename> ::= '[[' <ID> '<-' <ID> ']]'		*		PNrename	{ }	OK	gen(): gen 1st PNaction; ".rename("; gen 2nd PNaction; ")"
<exp> ::= ('(' <exp> ')' | <numvar>	*see <numvar> above*						
| '-' <exp> | <exp> <op> <exp>) <op> ::= ('+' | '-' | '*' | '/' | '==' | '<' | '>' | '<=' | '>=' | '!=')		*		PNop	>	OK	"("; gen left exp; op; gen right exp; ")"
Prep-time node substitution: new extracted subagent's <signature>			*	PNsigSub	{ }	>	prep(): note subagent no. in translation context gen(): NC
replaces complex <agent> subtree, refers to subagent	PNconst			PNconstSub	{ }	OK	default to PNconst::gen()

3.2.3 Simple Changes

Some of the operators of csp12 and CSPm are the same and required no changes to the translator (e.g., ‘;’, ‘+’, and ‘-‘, stayed the same). Others required simple token replacement (e.g., ‘::=’ became ‘=’, ‘=’ became ‘==’, etc.). Changes to the comment token in csp12 started as a simple replacement (i.e., ‘%’ became '--') but involved further development to handle CSPm’s multi-line comments.

CSPm does not use a statement terminator at the end of process definitions as csp12 does with ‘.’. CSPm allows identifiers, whether process names, channel names, or variable names, to be of any case beginning with an alphabetic character. Csp12 made a distinction between upper and lower case identifiers for processes, events, etc. Processes needed to begin with an upper case character. Actions needed to begin with a lower case character. Variables needed to begin with an underscore. Removing the ‘.’ and changing identifier rules affected the line numbers that were used to interleave generated code with CSP process definitions for improved readability. These two changes disrupted the line numbers passed to the ParseNode objects. In order to solve this problem a mid-rule action, linecheck, was introduced, that executed to determine the current line number. The CSPm source was then able to be cleanly interleaved with the generated C++ code in the output source file.

The external choice operator was not implemented with all the capabilities of the full CSPm operator. Our external choice operator only operates on prefix processes or processes in which we know they are prefix processes underneath. Consider the following example of an external choice with a conditional alternative:

```
MARATHON = MARATHON(0)
MARATHON(km) =
   if (km >= 40)
     then (finish -> STOP)
     else (run -> MARATHON(km+1))
   []
     quit -> STOP
```

Notice that either process resulting from the condition is of type prefix. When the ParseNode object PNifthen is constructed, it knows it is of the prefix form since it finds that both resulting processes are PNprefix objects, and this allows the choice operator to be translated.

3.2.4 Channel I/O

It was realized that by conscripting Datum objects to serve as containers for multiple data values, it was not necessary to implement a whole new mechanism in the framework suited for handling dot-delimited values and variables. The front-end was changed to disguise Datums as CSPm channel data. CSPm does not need the Datum names so the translator generates them internally from the channel name by prepending an "_". This is done in the bison file as the ParseNode PNdatum is being constructed.

In csp12 CSP++, matches between Datums were determined by comparing their pointers. When Datums created in UCFs to be passed back to the CSP++ backbone were compared to internally created Datums, the pointers would not be the same. The framework was changed slightly to compare more than just Datum pointers. Even if the pointers are not the same, the Datum names are still compared. Only if the names and the lengths of the Datums are the same do the Datums match.

3.2.5 Data Types

As mentioned above, experimental support for enumerated data types was implemented. In the following example,

```
datatype Number = zero | one | two
```

the name “Number” is added to a new symbol table for data types so that it cannot be redeclared in other `datatype` declarations. Also, each data type value (i.e., zero, one, and two) is registered in a data type value symbol table so that each value cannot be redeclared subsequently. CSP++ is still only set up to handle integers so, when the translator assigns the data type values as part of enumeration, the values are handled as integers internally. Since CSP++ programs have the option of outputting the traces of the system to stdout using the “-t” command line option, the data type values stored as integers are also printed as numbers rather than by name. For this and other reasons mentioned elsewhere, we have commented out `datatype` support until it can be more fully implemented.

3.3 Framework Changes

While some changes to CSP++ could be accomplished by modifying the translator alone, others required modifying the framework to produce CSPm-style semantics. Moreover, there were many new features such as multilevel synchronization that were needed to meet our goals of making CSP++ a more powerful tool. In the next section, we describe the changes made to the OOAF.

3.3.1 Parametric Changes

Just as the translator had some very simple but helpful changes, so did the framework. With the development of slightly larger case studies like the ATM, it was clear that the supported lengths for subscripts and datums needed to be increased. We increased both to 10 as the ATM case study needed to perform I/O with at least 7 data values. This was mostly a matter of altering some compile-time symbolic constants. The new values of these parameters are shown in Table 6 in section 3.5 (Restrictions and Limitations).

3.3.2 Subscripted Channels

To implement our policies established in section 3.1, we needed to allow Channels to have subscripts. In order to continue using the << and >> stream notation on Channels in the generated .cc file, there needed to be some way to specify subscripts as well. We chose the convention channelname(sub1,sub2,...,subn) << outputLit. This convention required that channel I/O become a two-phase operation. The first phase involved recording the subscripts in a new ActionRef object (containing information about an Action) allocated on the heap inside the new Channel::operator() function. That function wraps the new ActionRef in a Channel& reference returned for use in the second phase, where the operator<</>> functions execute the Action and then delete the ActionRef from the heap. The ActionRef created in the operator() function needed to be dynamically allocated on the heap rather than the stack so that it could be referenced by the operator<</>> functions after the return of the operator() function. The ActionRef::operator== needed to be changed to compare subscripts for any event instead of just Atomic events. The translator's bison grammar definition file needed to be changed to allow for parsing subscripted Channels. The translator's Symbol.h/.cc files

also needed to be changed to generate ActionRef object definitions that included the number of subscripts since, in csp12 CSP++, ActionRef definitions for Channels did not use a parameter for the number of subscripts.

3.3.3 Event Sets

As the framework executes the cspt-generated system, an environment stack is built up, comprised of the hiding and synchronization sets and the renaming lists of the currently executing Agent and its ancestors. As new subprocesses are created through composition, the environment stack "branches". In this way, the environment stack becomes an environment "tree" with the SYS process at the root. Before the current Agent can complete the execution of an Action, it first searches the environment stack to see if there are any "event sets" that would affect its behaviour. These event sets are not really "sets" in CSP++ but rather Env objects that are pushed onto the environment stack. The Env objects contain information concerning their purpose (synchronization, hiding, renaming) and which Action they refer to (via an ActionRef object). A synchronization Env object (EnvSync) is used to control synchronization for a specific ActionRef by registering which Action synchronizations are in progress, tracking which Actions are waiting to attempt synchronization, storing any data values to be communicated in the synchronization, as well as other synchronization details.

In csp12, every Action with a subscripted value had its own ActionRef. For example, 'a(1)' and 'a(2)' (i.e., the csp12 equivalent of CSPm's 'a.1' and 'a.2') each had their own ActionRef object. When these Actions were listed in event sets, individual ActionRef objects were created for each one. However, with the introduction of set closure in the

new CSP++, the number of ActionRef objects that would need to be created for each Action could be enormous. Consider a conservative example of an Atomic Action with three subscripts in the range {1..5}. The number of ActionRef objects needed would be 125! This would be very inefficient in CSP++ since most of those ActionRef objects would never even be used.

To avoid this explosion of ActionRef objects, we now create one ActionRef for each channel name, with the number of subscripts needed as a parameter to the ActionRef constructor.

The restriction of allowing only bare channel names in event sets permits hiding to achieve the set closure effect by hiding any Action that begins with the same channel name. For synchronization, the translator generates individual ActionRef objects for specific Actions, but we create them dynamically as they are needed. To accomplish this, only a synchronization set Env object (EnvSyncSet) is pushed on the environment at first. EnvSyncSet contains a list of dynamically created EnvSync objects. Only when an Action attempts to synchronize does a EnvSync, wrapping a specific ActionRef, get added to the environment stack (that is, unless the appropriate EnvSync is already in use—waiting for a synchronization). This keeps us from having to store ActionRef objects for all the possible expansions created by set closure. Both the specific EnvSync and ActionRef objects are removed after they are no longer being used for synchronization. However, if an Agent is waiting to synchronize because the EnvSync is already in use, the EnvSync and ActionRef must be kept until all the synchronizations are complete.

3.3.4 Multilevel Synchronization

Multilevel synchronization proved to be the most challenging feature to implement for the reengineered CSP++. Finding a solution that worked with external choice and channel I/O was also necessary. This section briefly describes the limitations that csp12 CSP++ had for synchronization and explains some of the solutions that were considered to make CSP++ compatible with CSPm.

For the csp12 version of CSP++, consider the process SYS ::= ((A||B)^{f}||C)^{f}. When ‘f’ occurs in A and B, each Agent searches the environment stack until it finds an ‘f’ in a EnvSync object. Either A or B returns as the “active party” and continues to search the stack for hiding or renaming. The active party is the last Agent to attempt to sync on a given EnvSync object. It is called “active” because it need not block. All other Agents are called “passive parties,” since they must block until the last party arrives at the rendezvous. Since csp12 CSP++ processes were restricted to synchronizing on the same level of the environment stack, if the active party encountered another ‘f’ on the stack for synchronization, it would outputs an error “action already taken for sync”. There is no such restriction in CSPm, where all processes in system with ‘f’ in their synchronization sets participate together for every occurrence of ‘f’.

A few different solutions were considered for supporting CSPm multilevel synchronization operational semantics without unduly degrading performance. One approach was to maintain a central “scoreboard” where Agents could post requests for synchronizations. This seemed like reasonable solution at first but was discarded when it became clear that maintaining a scoreboard separate from the environment stack would still require complete knowledge of the stack to properly enforce the CSPm semantics on

synchronization attempts. It made more sense to keep synchronization points in EnvSync objects as part of the environment stack. Another approach aimed to minimize the number of EnvSync objects for a given synchronization by having the uppermost EnvSync object contain all the information needed for the whole synchronization. However, since the number of synchronizing Agents for a given synchronization can be different at different times—that is, the system's process structure can grow or shrink dynamically—it was too difficult to determine how many Agents to wait for before completing the synchronization. The basic idea of our chosen solution was allowing the active party to attempt synchronizations again at higher levels of the environment stack. This solution will be discussed next.

By enabling the active party to attempt another synchronization higher up the stack, synchronizations between multiple parties on multiple levels are allowed. However, by allowing the active party to continue search and thereby having the chance of in turn becoming a passive party as it searches the stack, this clouds the definition of what an "active party" is. What we called the "active party" in CSP++ csp12 could now be termed "local active party," because the current Agent may become "passive" later if it is not the last party to attempt synchronization across all levels. This means that there may be different Agents that are the "local active party" at different times in the same synchronization. The Agent that is a local active party and is hidden or reaches the top of the stack is *the* active party that is responsible for the completion of the synchronization. Completion involves cleaning up any flags that were set during the synchronization, rolling back any other attempted synchronizations (if involved in choice), and ensuring each party receives the data (if channel I/O performed). Local active parties do not

complete synchronizations anymore. Instead of completing before a party returns as active, completion occurs after an Agent determines that it is *the* active party. An Agent determines that it is *the* active party when any of the following conditions are found:

- the Agent reaches the top of the stack (i.e., there are no more Env objects on the stack to be searched), or
- the Action is discovered to be hidden
- tracing is off and there are no more relevant EnvSync objects above. The first time a synchronization is completed for a given ActionRef, the highest EnvSync object's type is changed from EN_SYNC to EN_TOPSYNC. This indicates to Agents searching the stack subsequent to the first time that there are no more relevant EnvSync objects above to find. This is intended to save search time and increase performance. See section 5.5 to see the difference in performance this makes.

When the Agent determines that it is the active Agent, it finds, in the last EnvSync object it used (i.e., the top-level EnvSync), pointers to the last EnvSync objects used (if any) by the two Agents participating in the top-level synchronization. All EnvSync objects have pointers to the EnvSync objects below them. This facilitates recursive clean up on multiple levels of the stack. The EnvSync nodes also have pointers to the participating Agents in the synchronization so that the final agent can provide each Agent with the data value for the event undergoing synchronization.

If choice is involved in the synchronization, some Action executions may have to be delayed and saved in order to give other choice alternatives a chance to execute and

possibly complete a synchronization. The reexecute() function is used to revive a delayed and saved Action. It tries re-executing the Action, but needs to know if the Action's Agent had already searched to the top EnvSync node before the Action's execution was delayed. This information is provided as an argument to reexecute() and is obtained by loading the saved synchronization state with loadSync.

Figure 2 illustrates the way that CSP++ handles the execution of the following CSPm specification:

```
SYS = P [|{|c|}|] Q
P = c!2 -> SKIP
Q = R [|{c}|] S
R = c?x -> SKIP
S = c?x -> SKIP
```

Process P broadcasts the value 2 over channel 'c' to process Q, which is composed of processes R and S. They both receive the channel input and store its value in their respective 'x' variables. The trace of SYS would record a single 'c.2' event as the channel communication synchronizes among all three processes, P, R, and S. This is called "multilevel" synchronization because the participating processes are executing at different levels of the process structure.

Figure 2(a) shows the state of the environment stack before synchronization has begun. SYS and Q have both pushed an EnvSync object on the environment stack to represent the synchronization set {|c|}. The envelopes reserved for data transfers by each EnvSync object are initially empty as shown by the folder icons. Furthermore, these objects reserve three slots for each of the two parties (i.e., Agent objects) involved in synchronization at that level, as explained below:

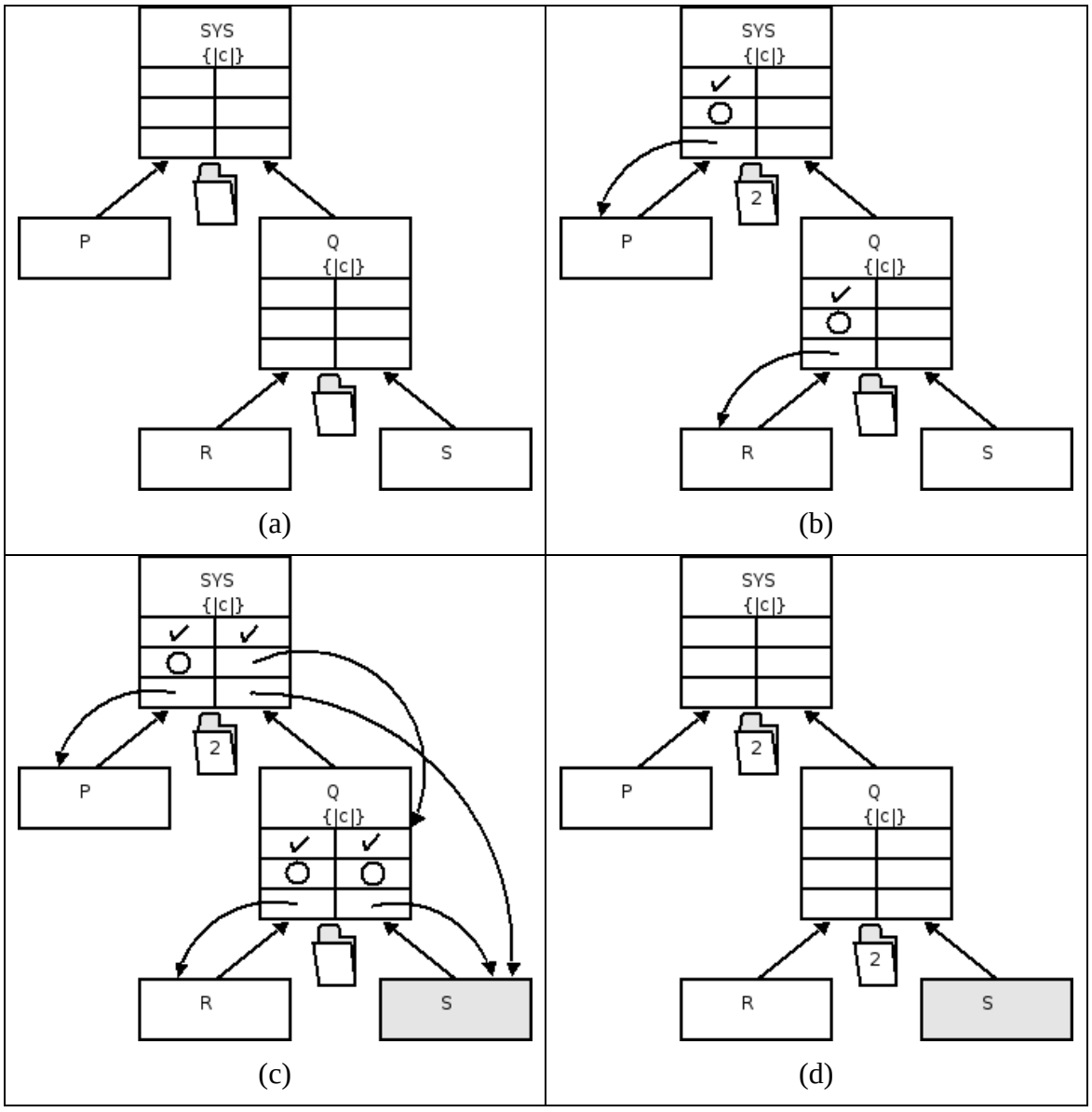


Figure 2 Multilevel Synchronization and Communication

- The first slot records whether or not a party has requested synchronization—a check mark in the figure indicates that synchronization has been requested.
- The second slot is used to dynamically construct a tree of all the EnvSync objects that are eventually found to be parties to the same synchronization. It stores a

pointer to the child EnvSync object below it in the tree (which is termed the "previous" EnvSync because the tree is built from the bottom up).

- The third slot points to the party that requested the synchronization.

In Figure 2(b), P has registered itself for synchronization on the event 'c.2', as shown by the checkmark in the first slot. There is no previous EnvSync object ("O" in the second slot) as yet. P records itself in the third slot as the party involved in the synchronization. The output value 2 is recorded in the envelope so that it can be communicated later on as the synchronization proceeds. P then sleeps until this synchronization is completed. Similarly, R has registered itself with the EnvSync object at Q. No value is recorded in that envelope because R is requesting input, not performing output. R also sleeps until this synchronization is completed.

In Figure 2(c), the other party for the Q synchronization, S, has arrived at the rendezvous. S has registered itself for synchronization, and, since the other party R has already arrived, S determines that it is the "local active" party, i.e., it is responsible for completing the synchronization at this level. Therefore, S searches up the stack to find any other matching EnvSync objects. It finds one at SYS, registers for synchronization at that level. It fills in the third slot with a pointer to itself as the requesting party, and inserts in the second slot a pointer to Q's EnvSync object for {|c|}, the child of SYS's EnvSync object for this synchronization. S finds once again that it is the local active party because P has already arrived. Finding that the EnvSync object's envelope at SYS is full, its contents '2' is copied by S. Then S continues searching up the stack and determines that no more matching EnvSync objects are found. At this point, S realizes

that it is *the* active party and that it is responsible for the completion of the synchronization throughout the tree of EnvSync objects that has been built up.

S effects a bottom-up cleanup of the synchronization tree by calling cleanup() on the topmost EnvSync object, the one at SYS, with the value of the envelope as an argument. The cleanup() function is recursive, invoking itself on any "previous" pointers before cleaning up at its own level. "Cleaning up" means canceling other choices that were not taken, resetting flags, passing channel data to any inputting party, and waking sleeping Agents. The result of the cleanup is shown in Figure 2(d): The slots are cleared, and the envelope data (i.e., 2) has been copied to R and S. Process S maintains control throughout the cleanup, and continues execution with S's next event.

3.4 Summary of Restrictions and Supported Syntax

Since CSPm is a dialect suited for more than just synthesis, we restrict CSP++ to accept only a synthesizable subset of CSPm. Some restrictions are also enforced to reduce the complexity of CSP++ and increase performance. These restrictions do not limit the power of CSP++ significantly because many restricted operations can be done in other ways that are supported in CSP++. In the next section, the restrictions and limitations made to CSP++ will be summarized.

3.5 Restrictions and Limitations

The following table shows the restrictions that are in place in the current version of CSP++.

Table 5 Restrictions in current CSP++

AREA	RESTRICTION	CONSEQUENCE
Action	If an external routine is linked to an Action, the Action cannot also be used for sync.	Make sure that internal and external Actions are distinguished by name in the CSPm specification. If an Action is needed externally and internally, write something like *eventname* for external use and *eventname_i* for internal use.
Agent	All definitions of the same-named agent must have the same number of non-overlapping arguments. Constant arguments can only be integers.	One cannot define, say both X and X(i), nor X() and X(i). Instead, define only X(i) and start by testing i for 0.
PNinput/ PNatomic/ PNchannel	Actions must always appear with the same number of subscripts except in sets	Values must be explicitly written out. For example, c?x cannot synchronize with c!1.2 nor can c?x.y synchronize with c!2.
cspm.y	Only bare channel names allowed in sets and closure sets	To synchronize with some events starting with a channel name and not others (the purpose of allowing subscripts in sets), the event can be defined differently (e.g., instead of ‘a.1’, try ‘a_1’).
PNatomic/ PNchannel	Atomic Actions cannot be mixed with Channels in CSP++	This implies that in CSP++, c.1 cannot synchronize with c?x, for example, even though CSPm allows it. This would be caught at translation time.
cspm.y	Channel I/O must use exactly one communication field operator ('?' or '!')	Events like c?x!y are not allowed. Mixed I/O events are disallowed. Events like c?x?y can just as well be rewritten c?x.y.
Action	Cannot hide input without corresponding output	This prevents the specification from becoming nondeterministic.
cspm.lex	declarations must not be split across more than one line unless through word wrap (i.e. declarations must not have carriage returns).	For example, one could not write channel e1, -- event 1 e2 -- event 2
Agent	Only one output per synchronization	For example, c!2 will not synchronize with c!2. CSP++ synchronizes on the

Table 5 Restrictions in current CSP++

AREA	RESTRICTION	CONSEQUENCE
		channel name and would try to synchronize c!2 and c!3. CSP++ would then flag this as a runtime error even though CSPm would not try to synchronize them at all since they are different events.

For quick reference, the following table shows the locations of numerical limitations that are in place for the current version of CSP++. The limits for constants are defined in Limits.h.

Table 6 Locations of Limitations

CONSTANT	LIMITATION (Max. no.)	IMPACT OF INCREASING
AG_ARGS	AgentProc arguments (10)	More storage for array of Lits, and more calls to Lit constructors/destructors when Agents start/terminate
AG_COMPOSE	Agents that can be composed (8)	More storage for syncFlags bit strings in EnvSync object (negligible)
AT_SUBS	subscripts (10)	Code more arguments for ActionRef constructor, Atomic::operator() and Channel::operator().
none, see Lit.h	Datum subscripts (10)	Code more DATUM_n macros.

This chapter presented the theoretical and technical discussion of the changes needed in the reengineering of CSP++ for CSPm tool conformance. The next chapter presents an

Automated Teller case study to demonstrate the development of a system using CSP++ and the selective formalism design flow.

Chapter 4

Automated Teller Case Study

The disk server subsystem (DSS) was a small proof-of-concept case study that demonstrated many of the features of the of the csp12 version of CSP++ but did not attempt to use the selective formalism design flow. In this chapter, we will demonstrate this design flow with a new Automated Teller Machine (ATM) case study developed for the new CSPm version of CSP++ and the CSPm verification tools. With the support of commercial tools, we are now in a better position to understand how to use CSPm in the selective formalism design flow. The newly reengineered CSP++ needed to be tested with case studies and have its performance measured. The integration of UCFs in CSP++ needed to be explored further in the ATM case study. Since the original DSS case study's UCFs were limited to simple print statements, UCF input and complex I/O needed to be tested and thought through more extensively. There are still many issues to be considered in the proper integration of UCFs to CSP++, and as they are increasingly used in CSP++ case studies, the areas for improvement will become clearer.

The ATM case study implements a small software system based on the requirements documents from a full object-oriented design by Professor R. Bjork at Gordon College [An Example of Object-Oriented Design: An ATM Simulation] who followed all the steps of OO methodology leading up to a final Java implementation. Neither our ATM case study nor Bjork's Java-based implementation are "real" systems, but provide reasonably detailed simulations of real systems. At the moment, there are a number of

challenges to face when trying to find a suitable case study. Below are some of those reasons:

- Systems should be complicated enough to benefit from formalism and verification
- Detailed design documents for “real” systems are not readily available
- Limited knowledge of specific systems in industry
- CSP++ has only been ported to personal computer systems
- Financial cost of hardware accessories

The ATM was complex enough to benefit from verification. We provide examples of verification in section 4.3. Although Bjork’s ATM was not a “real” system, it was more complex than the DSS and also came with many helpful design documents that explained the workings of the system. The ATM simulation case study also incurred no financial cost.

Of Bjork’s requirements documents, we use the UML Statecharts, use case, and functional test case documents as steppingstones for our ATM design in CSP++. Bjork’s Statecharts for the ATM can be found in Appendix A. The use case diagram, below, shows the various actors that interact with the ATM system.

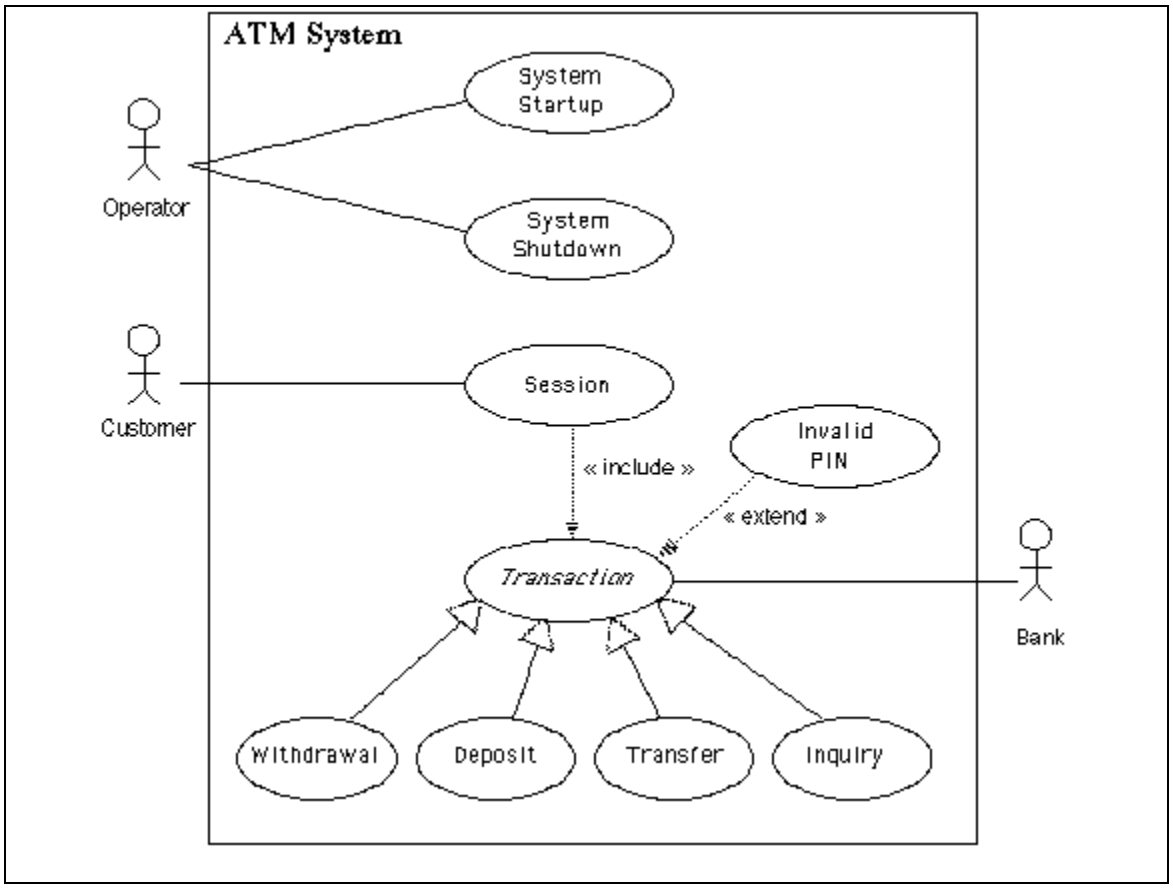


Figure 3 ATM Use Case Diagram

Bjork's implementation is a standalone Java application where actors are simulated within the application. Our CSP++ ATM communicates via sockets with an external bank program that in turn accesses a MySQL database. The rest of this chapter walks through the selective formalism design flow for building the ATM system. We begin by explaining in general how systems can be designed in CSPm.

4.1 CSP in the Design Phase

How is a system to be designed using CSPm? Textbooks teach the constructs of the CSP formalism, but do little to show how CSP can actually be used to model a system or be

incorporated in a repeatable software engineering process [Schneider 2000, Roscoe 1998, Hinchey, Jarvis 1995]. We have identified patterns of using CSP that amount to designing four complementary models:

1) Functional Model
2) Environmental Model
3) Constraints Model
4) Implementation Model

These four models have been published in [Doxsee, Gardner 2005a, Doxsee, Gardner 2005b]. In this section, we explain the purpose of the four models and how their CSPm code is derived from Bjork's requirements to target CSP++ synthesis.

1) Functional Model

This model captures the overall behaviour of the system in CSP with processes engaging in named events. In the ATM case study, the CSPm that resulted from the translation of the UML Statecharts served as the functional model role in our design flow.

2) Environmental Model

The environmental model simulates those entities that interact with the main system. These entities are modeled as processes that provide stimulus for the functional model. Once the system is ready to be deployed, the environmental processes are removed to leave the synthesized system to interact directly with its real environment via UCFs. The ATM case study had four main entities—three of which were environmental processes. The first entity, the ATM process, was itself made up of subprocesses. The three environmental processes represent the other systems that the ATM interacts with: an

OPERATOR, BANK, and CLIENT. The OPERATOR process simulates an operator who turns the ATM on or off and sets the amount of cash it holds. The BANK serves as the provider and maintainer of account information necessary for the validation of transactions such as the account balance or the PIN number. Finally, a CLIENT must be able to interact with the ATM performing transactions of different sorts.

3) Constraints Model

Other CSPm processes may optionally be placed alongside the functional model processes to limit or constrain the sequence of events that are permitted to occur. There may be many possible traces that result from even a simple specification. A constraining process can force certain named events to occur before others, for example. One way to write CSPm specifications is to incrementally compose several processes in order to increasingly constrain the specification until a proper system model results. The train crossing example below shows how constraints model processes are combined to form a specification:

```
channel open, close, arriving, gone
GATE = close -> open -> GATE
TRAIN = arriving -> gone -> TRAIN
SIGNALGATE = arriving -> close -> SIGNALGATE
   [] gone -> open -> SIGNALGATE
CROSSING = (GATE ||| TRAIN) [|{open,close,arriving,gone}|] SIGNALGATE
```

Here, the GATE and TRAIN processes describe the behaviour of the gate and the train, respectively. The SIGNALGATE process is a constraints model process that ties the events of GATE and TRAIN together to specify their combined behaviour. The resulting trace is <arriving,close,gone,open> repeated any number of times. The ATM specification did not need a constraints model.

4) Implementation Model

The functional model is not adequate to fill in all the details of the specification. Writing an implementation model involves adding extra processes or events to the functional model to complete the specification. Verification will reveal whether or not the specification is consistent and valid. For example, the ATM Statecharts did not contain the details for handling invalid PIN entries, so CSPm implementation model processes were added along side other processes to complete these important details.

In the sections that follow, we will see how each of these four models work together to design a system with CSPm. In the next section, we demonstrate how the functional model can be derived from Statechart diagrams of the system.

4.2 Writing the CSPm Specification

We have already seen a simple example in the previous section of how a train crossing can be specified by composing several constraints model processes. In this section, we present two ways that CSPm functional models can be derived from Statecharts, and two ways of modeling and handling variables and data. Some of these approaches may be preferred for their convenience and others for their performance (see section 5.3). Finally, we choose an approach for modeling the ATM and discuss how it is applied.

4.2.1 Deriving CSPm from Statecharts

The following are two techniques for arriving at specifications from Statecharts or Hierarchical Concurrent Finite State Machines (HCFSMs). We call the approaches “flat” and “hierarchy”:

1) Flat: Collapse the state machine hierarchy into one single state machine. When the resulting CSP is synthesized and executed, only one thread is created.

2) Hierarchy: Keep the hierarchy from multiple state machines, and mimic it with a hierarchy of processes that transfer control to other subprocesses by synchronizing on common event names. This has the effect of a state machine transmitting control to the next state machine while still holding its own state.

When deriving CSP specs from Statecharts, events for synchronization with the environment may not translate one-to-one in CSPm. Since events destined to be replaced by UCFs are unable to be also used for internal synchronization in the current version of CSP++, events must occasionally be repeated (under a modified name) so that one can be replaced by a UCF and the other used in synchronization with other processes. We suggest that the following naming convention be used for processes needed for UCFs and synchronization:

- Event replaced by UCF: eventname
- Event for internal synchronization: eventname_i

4.2.2 Handling Data and Variables

There are different ways to handle data and variables that are required by more than one process. Consider the scope of the data in the following three process definitions:

```
P = Q(2)
Q(y) = c?x -> R
R = d -> SKIP
```

Process P passes the value 2 to Q(y) so that y is bound to the value 2 for all of Q(2). Once the variable ‘x’ in ‘c?x’ is bound to a value, ‘x’ continues to have that value for all of

Q(2). However, when Q(2) continues as R, both the value for ‘x’ and the value for ‘y’ are not visible to the process R. That is to say that data is not transmitted automatically from process to process. Here are two means of transmitting such data:

1) Parameters: Variables can be passed from process to process by means of process parameters. In the example below, the READINGCARD process could have been implemented to pass the card number ‘c’ as a parameter to READINPIN(c).

```
READINCARD = readcard?c -> READINGPIN(c)
  [] badcard -> EJECT
```

   Passing parameters may make the program more difficult to understand if long lists of parameters are passed along in large programs.

2) Global variables: A process can be dedicated to setting and getting a value for an individual. For example, a variable for PIN could be provided via processes like these:

```
PINi = setpin?x -> PIN(x)
PIN(val) = setpin?x -> PIN(x)
    [] getpin!val -> PIN(val)
```

   This way a process could store a PIN value by synchronizing with the PINi process’s setpin?x event. The value could later be retrieved by synchronizing with the getpin!x event. By using these so-called global variables, parameters need not be passed along throughout the specification.

Long lists of parameters are cumbersome and their values may not be needed until much later in the specification. On the other hand, depending on the efficiency of the underlying thread model for creating threads, there may be some disadvantage to having a thread created for each variable with regards to execution speed.

4.2.3 Choosing an Approach for Modeling the ATM in CSPm

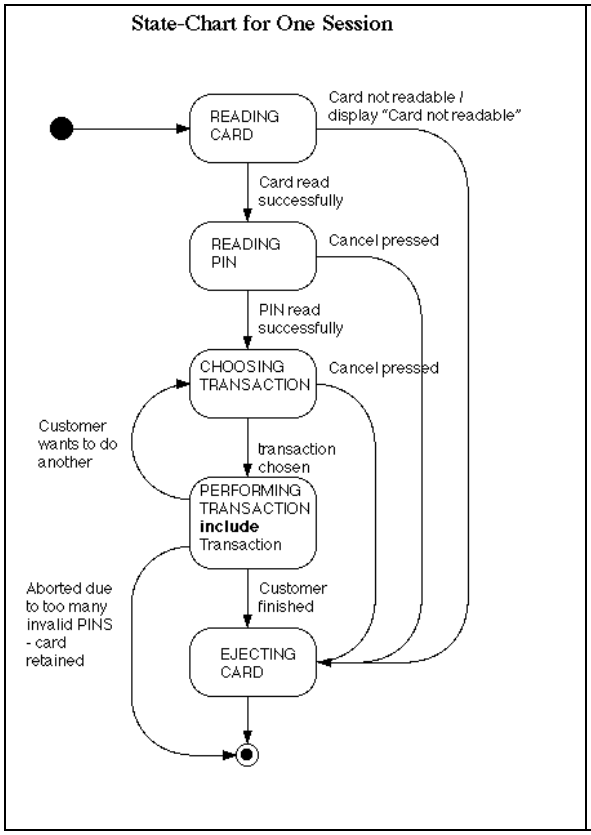
Thus far, experience has demonstrated that Statecharts or finite state machine models of the system requirements are valuable in the design of CSP++ systems. State machine representations lend themselves well to being modeled in CSPm, where states become processes and transitions become events. The CSPm derived from Statecharts would be considered to form the functional model of the specification. If the state machines are hierarchical concurrent, as are the state machines of the ATM example (see Appendix A), then each state machine can be its own process (one thread) that synchronizes on key common events with other processes. A large, complex state machine can often be broken down into hierarchical concurrent state machines [Vahid, Givargis 2002]. Breaking the design into parts keeps the design simple. In contrast, viewing the system as a giant state machine can complicate one's understanding of the system. State machines are an easy way to visualize the movement of the system state. Tools could be developed that translate state machines into CSPm specifications to simplify the specification process. Deriving CSPm from a single monolithic state machine minimizes CSP++'s thread usage, thereby increasing runtime performance.

As can be seen in the ATM use case diagram in Figure 3 earlier this chapter, there are three processes in the environmental model (BANK, CLIENT, and OPERATOR) that interact with the main ATM system. These are all composed to make up the entire system (SYS) as can be seen in the following CSPm code excerpt:

```
SYS = ((ATM
   [|{|banksend,bankstatus, commit|}|] BANK)
      [|{|insertcard,readcard,readpin,choose,getacct,getamnt,dispense,
           again,badcard,cancel|}|] CLIENT )
           [|{|on,machcash,off|}|] OPERATOR
```

To implement the ATM alone for synthesis, the bank, client, and operator would be removed and the channel inputs and outputs of the ATM would interface with UCFs that accept button pushes, or even provide network connections.

The ATM process is itself composed of a few subsystems that communicate together and work as illustrated by the state machine designs in Appendix A. The ATM has an overall behaviour. Within the overall behaviour of the ATM, a session with a client can occur. The session may involve a number of transactions. Figure 4 shows the correspondence between the session subsystems and the CSPm code derived from it.




```
SESSION = insertcard_i ->
READINGCARD

READINGCARD = readcard?c ->
(cardset!c -> READINGPIN)
[] badcard -> EJECT

READINGPIN = readpin?p ->
(pinset!p -> CHOOSING)
[] cancel -> EJECT

CHOOSING = choose?menu ->
(choose_i!menu -> TRANS)
[] cancel -> EJECT

TRANS = endtrans -> EJECT
[] anothertrans -> CHOOSING
[] holdingcard -> DONE

EJECT = ejectcard -> DONE

DONE = sessiondone -> SESSION
```

Figure 4 Correspondence between Statecharts and CSPm

The CSPm in the above figure was derived partially from the Session Statechart by changing state names to process names and transitions to events. Multiple transitions from a single state are modeled in CSPm as events participating in external choice. Transitions that represent some type of I/O are modeled with channel I/O (e.g., the “PIN read successfully” transition is ‘readpin?p’ in CSPm).

For the ATM, we chose to use the “hierarchy” approach for deriving CSPm, and the “global variables” approach for modeling and handling variables and data. The whole Session Statechart could also have been represented by a single SESSION process, but breaking it apart into different processes (i.e., SESSION, READINGCARD, READINGPIN, etc.) yields a closer correspondence between CSPm and Statecharts. Also, the many processes are handled efficiently in CSP++ as only one thread is created for them all.

Since we chose the “hierarchy” approach, the three Statecharts in Appendix A become the OVERALL, SESSION, and TRANSACTION processes in synchronized composition. SESSION is linked with the OVERALL parent system and the TRANSACTION subsystem through synchronization on common events. insertcard_i is used for internal synchronization with the OVERALL process, and choose_i is used for internal synchronization with the TRANSACTION process. All the CSPm code is listed in Appendix B.1.

4.3 Verification

Ease of verification is one of the greatest benefits resulting from reengineering CSP++ for CSPm. It is now possible to incorporate commercial tools smoothly in the selective

formalism design flow. The commercial verification tool, FDR2, makes it possible to detect problems with the CSPm specification including:

- Deadlock: The situation where the system is no longer making progress because every process is waiting on another other process.
- Livelock: The situation where processes enter into an endless sequence of interactions with themselves to the exclusion of any external interaction.
- Nondeterminism: The situation where program's execution cannot be predicted by prior events.

As well as detecting problems like the ones above, verification allows trace refinement and failures refinement assertions to be written to expose more subtle problems in the specification (see section 2.1.6). For these subtle problems, it is best to have at least one "CSP guru" for selective formalism design with CSP++ who knows how to "ask the right questions" of the specification to complement other "non-guru" developers who write CSPm or UCFs. Some examples of these more complicated verification assertions are given in the rest of this section. If wrong questions are asked, those verifying the specification may be mislead into believing that specifications are fine when they are not. Even without a "guru," the selective formalism design flow still has the benefits of automatic code generation from specifications and the ability to perform some verifications automatically.

Note that the verification phase of the design flow requires the environmental model. The environmental model is based on the system designer's assumptions of how the real environment of the system will operate and interact with the system. We will now show

some of the ways in which verification can be used in the selective formalism design flow as well as provide advice for keeping the state space of the specification small.

4.3.1 Basics of Verification

Before verification, it is often wise to first use the Type Checker utility to detect data type errors within the CSP specification. Other specification problems may remain after using the Type Checker, but the specification will be in better shape for verification with FDR2 and simulation with ProBE. All of these tools accept *.csp files in which the designer writes the CSPm specification.

Once the data type and syntax errors have been eliminated, the specification is ready for verification. FDR2 provides some automatic checking capabilities:

```
assert ATM :[deadlock free [F]]
assert ATM :[livelock free [F]]
assert ATM :[deterministic [F]]
```

As was mentioned earlier, more realistic and useful verification requires “asking the right questions” of the tool using carefully thought out assertions. For the ATM case study, Bjork’s requirements documents provided functional test cases that proved to be of great value for “asking the right questions” for the verification of the CSPm specification.

4.3.2 Trace Refinement

By using trace refinement in FDR2, it can be shown that functional test cases are satisfied for safety. A system demonstrates its safety if it does not do more than it is intended to do. One functional test case requires the demonstration that a client’s card will indeed be held after entering an invalid PIN three times in a row. Verification for the same functional test case may be done in a number of ways. Our approach began with

determining whether or not the three invalidPIN events happen within the same transaction attempt. If they do not, holding the card does not apply. The end of a transaction (normal or otherwise) is indicated by the ‘again’ event. So, in order to have the card held, we must receive three invalidPIN events before the user is given the option to, ‘again’, try another transaction. Below is an assertion written in CSPm for FDR2 using the `assert` command.

```
assert ATM \ diff(Events,{|invalidPIN, again, holdingcard|})
   [T= invalidPIN -> invalidPIN -> invalidPIN -> holdingcard -> STOP
```

By hiding all events except for those we are interested in from the ATM (using the set difference operator, `diff`, and the `Events` set to which all defined events belong), we can ensure that the card will indeed be held after three invalidPIN events in a row. This assertions should succeed. Notice that if we changed the trace portion of the above assertion to the following

```
[T= invalidPIN -> invalidPIN -> invalidPIN -> again.1 -> STOP
```

it would fail for safety because ‘again.1’ (the 1 signifying ‘true’) is more than the system can do after three invalidPIN events.

4.3.3 Failures Refinement

Failures refinement is used to show liveness (i.e., that a specification must continue to do what it was intended to do). Liveness is different from avoiding livelock because a livelock-free specification may still do something it was not intended to do. A failures refinement example comes from the System Startup functional test case where we want to prove that the ATM must continue to allow the “on” switch, a request for the initial cash amount, followed by the “off” switch. That is to say that failures of the ATM should

be a subset of a specification that repeatedly performs ‘on’, ‘machine.x’, and ‘off’. Such an assertion can be written in the following way:

```
P = on -> machcash?x -> off -> P
Q = ATM \ diff(Events,{|on, off, machcash|})
assert P [F= Q
```

The ATM satisfies the above liveness specification. Many other properties can also be verified by writing similar assertions and running them in FDR2.

4.3.4 State Space

One known difficulty with verification tools in general is the problem of state space explosion. Real-life values for channels may make up a very large set. Values for channels may be as simple as booleans (two possibilities) but may be as large as integer sets (account balance, for example, which has any number of possibilities). If a channel has as few as 3 values associated with it and the number of possible entities for each value is, say, 5, then there are 5^3 or (125) states for that one channel. Commercial verification tools may implement clever heuristics to avoid searching all the states, but still must go through the memory- and CPU-consuming processes of searching an potentially enormous number of states.

Ranges of values needed to be limited to sets of size 2 or 3 to keep the state space from exploding in FDR2 or ProBE and locking all the resources on the PC. This is not a problem that stems from a failure to set realistic boundaries on values but one based on the realities of formal verification where specifications must be fully and mathematically explored. For this reason, we often had to limit our value ranges substantially for formal verification.

Even though a specification writer might want to have a large number of possible values for a variable, it is often unnecessary and problematic to define large sets. CSP++ ignores channel declarations so if defined ranges are small for verification purposes, the resulting C++ code will not be limited.

4.4 Synthesizing C++ and Integrating UCFs

Once the verification phase of the selective formalism design flow is complete, the system can be synthesized. The root directory of the system's source files should include the *.csp specification file, any other .cc or .h files for UCFs, and a Makefile. The Makefile should be set up to translate the *.csp file using cspt and to compile and link the resulting .cc file with the compiled UCF files. The cspt-generated C++ code is easily debuggable in GDB on Unix systems where breakpoints can be set and variable inspection can be performed. The code follows the format of the CSPm specification quite closely making it easy to follow the chain of events and isolate problems.

The system can be synthesized for three different purposes:

- With the Environmental Model
- Without the Environmental Model
- With UCF integration

These three ways of doing synthesis for CSP++ systems will now be discussed.

1) With the Environmental Model

By keeping the environmental model processes in the CSPm specification for synthesis, the designer can build a system in which the traces of the system can be observed as they

are displayed on stdout. Run-time printing of traces is activated by using the "-t" command line option on the linked executable program. This way of synthesizing requires no external input as the environmental model processes in the system simulate each event.

To see how cspt translates a CSPm specification, consider the following CSPm fragment from the ATM case study:

```
READINGPIN = readpin?p -> (pinset!p -> CHOOSING)
[] cancel -> EJECT
```

Here, the READINGPIN process offers a choice between the event 'readingpin?p' and 'cancel'. If 'readingpin?x' is provided with input, the PIN number is stored using the "global variable" technique with the 'pinset!p' event, and the process continues as the CHOOSING process. If 'cancel' is executed, the process continues at the EJECT process that eventually ejects the card. The corresponding cspt-generated code for the READINPIN process is found below:

```
Channel readpin("readpin", readpin_p);
AGENTPROC( READINGPIN_ )
FreeVar p;
   Agent::startDChoice( 2 );
      readpin >> p;
      cancel();
   switch ( Agent::whichDChoice() ) {
      case 0: {
         pinset << p;
         CHAIN0( CHOOSING_ ); }
      default: {
         CHAIN0( EJECT_ ); }
   }
}
```

The cspt translator generates C++ code that employs the use of classes and functions defined in the .h files of the OOAF. The C++ code is similar to the CSPm source and is quite readable. The C++ begins by defining a Channel named 'readpin'. AGENTPROC

is a macro that begins the definition of the function for the READINPIN process. The FreeVar variable ‘p’ is declared so that it can be used throughout the function for input with `readpin>>p` and output with `pinset<<p`. Before input and output, the function begins a deterministic choice between the two alternatives of reading the PIN or canceling the session. After starting the execution of both alternatives, we find which deterministic choice succeeded and act accordingly. If the PIN was read, we output it to be stored in another processes and use CHAIN macro to continue execution as the function implementing the CSPm CHOOSING process. If the session was canceled, we continue execution as the function implementing the CSPm EJECT process.

When the environmental model processes are left in the CSPm specification, the input on the readpin>>p Channel is obtained from a corresponding output (e.g., readpin<<1234) in the function implementing the environmental model process for a CLIENT.

2) Without the Environmental Model

If the environmental model processes are removed from the CSPm specification, functions such as readpin>>p perform a default action: Channel input reads from stdin and Channel output prints to stdout. When the system is synthesized, built, and executed, the input replacing the outputs from the environmental model processes can be provided by the person executing the program through stdin. This allows the designer to interact with the system via the console.

3) With UCF integration

In order to allow events to act in custom (not default) ways with the real target environment, the environmental model processes are commented out and their events are replaced by user-coded functions. In the ATM case study, we used all the supported types of UCF channel communication (i.e. single integer I/O and multiple integer I/O).

One of the best indications in determining if an event is a candidate for a UCF is if it was designed to communicate with an event in the environmental model. One of those events in the ATM was ‘readpin?x’. Earlier this section, the C++ code for the READINGPIN process was presented. Below is a UCF that can replace the functionality of the readpin>>p C++ function. Its function is to obtain the client’s PIN and return it to the CSPm specification via the status argument.

```
void readpin_chanInput( ActionType t,ActionRef* a,Var* status,Lit* l)
{
   int pinnumber;
   cout << "Welcome to the CSP++ ATM" << endl;
   cout << "Please enter your PIN -> ";
   cin >> pinnumber;
   *status = Lit(pinnumber); // store input
}
```

In order to link the `readpin>>p` function in the cspt-generated C++ file to the external UCF it should be compiled with “-Dreadpin_p=readpin_chanInput”. In this way, the extraction (>>) operator of the Channel object readpin will use the externally linked readpin_chanInput function rather than the default Channel input behaviour.

There is still much more to research about UCF integration. If a CSPm specification has, through verification, been shown to behave properly, then the specification will behave properly. However, if, through UCF integration, invalid data is input into the synthesized formal backbone then one can no longer rely on the verified properties of the

system. At some point, UCF input must be validated. This could be done in the OOAF, the UCF, and/or the CSPm specification. Since the translator ignores type declarations, the OOAF does not know what is valid and what is not and therefore cannot validate the data. We will now look at the remaining options.

If the specification needs valid data from the UCF, there must be opportunity for the data to be reentered. Since the UCFs for input replace a single input event, the UCF should not input data more than once per call to reflect the behaviour of the CSPm specification. Suppose the CSPm specification extended the range of the data type to include an error flag. The UCF developer could make sure the data is valid by returning the error flag to the formal backbone and having the CSPm call the UCF-replaced function again. This solution requires validation efforts in both the CSPm and the UCF.

If the CSPm specification performed the validation itself, depending on what needs validating, it could be quite a cumbersome and unappealing solution. However, for simple validation, it may be simpler than using a UCF/CSPm combination solution. Until a new solution is discovered, we recommend using CSPm validation for simple cases and UCF/CSPm validation for more messy validation.

One question we had to address with regards to UCF integration was if and how UCFs could communicate modified data back to the CSP++ backbone. The danger of allowing UCFs to communicate with each other “behind the back” of the formally verified backbone is that it may potentially break the formalism. Properties that were verified may no longer hold if such communication takes place. A CSP++ rule that we have established is that *interprocess* communication must be performed strictly via CSP channels. User procedures can safely communicate with one another as long as they are

only ever invoked by the same process [Gardner 2000]. At the moment, we have no way to enforce this rule leaving the responsibility with specification designers and C++ programmers to follow this convention.

The new ATM case study provides another system with which to put CSP++ to the test. In the next chapter, this and the DSS case study will be used to evaluate the performance of CSP++.

Chapter 5

Performance Metrics

In this chapter we set out to determine the change in performance of CSP++ since it was last measured in the original work on CSP++ [Gardner 2000]. The changes outlined in the preceding chapters, including the change of the underlying thread model, were substantial and it remains to be shown whether or not CSP++ has maintained a competitive performance. Such an investigation also uncovers future work opportunities for optimization in those areas of CSP++ that appear to be slow.

Understanding a short version history of CSP++ highlights some of the changes that would affect performance:

- V2.1: original work based on LinuxThreads; timing/memory measurements given in [Gardner 2000]
- V3.0: first version based on Pth; no measurements were made
- V3.1: bug fixes, and source compatibility with gcc-3
- V4.0: first version with CSPm translator, but still csp12 framework semantics
- V4.1: full compliance with CSPm

To compare the performance of V4.1 with the performance of V2.1, they must be compared on a level playing field. All tests were run a 1.5 GHz Pentium M with 512 Mb of memory, running either Fedora Core 3, for CSP++, or Windows XP, for Rational Rose RealTime (RRRT).

The g++ compiler used for the CSP++ tests was gcc-3.4.4 with -O2 optimization. For the RRRT tests, the MS Visual C++ 6.0 compiler was used.

LinuxThreads tests were based on the glibc-2.3.5-.fc3.1 implementation of LinuxThreads, and the GNU Pth version is 4.0.0.

The CSP++ applications were run without the tracing option "-t" or the idle check option "-i" (that causes a dump after 2 seconds of inactivity). The applications were passed the quick exit option "-q" to finish without a dump when the system executes STOP.

Each test was run six times with the average of the last five runs being used for the recorded time in order to not count the effect of paging on the first run. Times were recorded using the Linux 'time' command. The total times for each run are the total of the system time and user time. The largest standard deviation for any group of five runs was 0.07 but the average standard deviation for all of the timing tests was only 0.02.

In this chapter we will investigate the following questions:

- *What effect does Pth have?* Recent versions of CSP++ are based on this third-party POSIX-compliant threads package.
- *What effect does static linking have?* Dynamic and static linking are known to affect performance.
- *What effect do the modifications have?* Comparing V3.0 with V4.1 will reveal how much the reengineering changes affected CSP++'s performance.

- *How does specification structure matter?* The ATM was specified in many different ways. It would be interesting to know what difference they made in performance.
- *Does the isTop feature reduce run time?* This feature was intended to cut the search time for the environment stack.

Finally, we also measure the sizes of CSP++ executables.

5.1 The Effect of Pth

The V2.1 timing tests in the original work were done using the preemptible kernel-space LinuxThreads implementation of Pthread that came with the Red Hat 6.2 distribution. Since then, the thread model was changed to the nonpreemptible user-space GNU Pth implementation of Pthread. Before measuring the effect of the latest modifications to CSP++, the effect of this change must be measured. In version 3.0 of CSP++, the thread model was changed to Pth. Without changing the CSP++ code, the threading model could be swapped to compare CSP++ with Pth versus CSP++ with LinuxThreads.

As the DSS case study was used in the performance testing for V4.1, it has become a benchmark for measuring CSP++'s performance. There were three different specification variations that were used as tests:

(1) 20,000 disk accesses in 20,000 process creations

```
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|}
```

(2) 20,000 disk accesses in 2 process creations

```
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|}
```

(3) 10,000 disk accesses; same as (1) with `Test(5000)`

C(1) and C(2) represent client processes that make requests to the disk server and receive acknowledgements. Test (2), with 20000 disk accesses and 2 process creations, is the one used in later tests unless noted otherwise. In all tests, the 'dint' and 'dio' events are hidden ('\' operator) so they are not output to **stdout** by default, which would simply inflate the execution time to no purpose.

In this section, we recreate the comparison from the original performance tests, change the thread library to Pth, and then run the same tests again. In order to compare the fastest times for Pth with the fastest times of LinuxThreads, we use static linking for Pth and dynamic linking for LinuxThreads. The results are given in Table 7 in the form "user time + system time = total time".

Table 7 Total Time (Seconds) for Disk Accesses with Pth and LinuxThreads, V3.0

	Test (1)	Test (2)	Test (3)
GNU Pth	29.21 + 5.49 = 34.7	22.13 + 4.32 = 26.45	11.06 + 2.17 = 13.23
LinuxThreads	0.92 + 1.22 = 2.14	1.17 + 1.23 = 2.4	0.58 + 0.61 = 1.19

From the test results we can make at least three observations.

1. LinuxThreads performs about an order of magnitude faster than GNU Pth.

2. There is a noticeable difference between 2 process creations and 20000 process creations.

3. Whether 2 or 20000 process creations is faster depends on the thread package.

Analysis:

The adoption of GNU Pth in CSP++ alone counts for a slowdown of about an order of magnitude. Clearly Pth is a source of surprising inefficiency.

5.2 Static vs. Dynamic Linking in V3.0

We thought it was worthwhile to see how static linking affected the performance of CSP++. Using the Linux 'ldd' command, it was determined that the following libraries are linked in dynamically: libpthread.so.20, libstdc++.so.5, libm.so.6, libgcc_s.so.1, libc.so.6, and /lib/ld-linux.so.2. Table 8 shows how static linking affects the performance of the DSS with Pth and LinuxThreads under version 3.0 of CSP++. From now on, the default test for the DSS will be 20000 disk accesses with 2 process creations.

Table 8 Total Time (Seconds) for 20000 Disk Accesses, V3.0

	Static	Dynamic
GNU Pth	29.21 + 5.49 = 34.7	50.15 + 5.44 = 55.6
LinuxThreads	1 + 2.17 = 3.17	0.92 + 1.22 = 2.14

We can observe that static linking speeds up execution time by about 38% when using GNU Pth and slows down execution time by about 48% when using LinuxThreads.

5.3 The Effect of Specification Structure

Conventional programming languages have “best practices” for writing code to make it more efficient. Does this concept apply to formal specifications written in CSPm? Since CSPm specifications are translated for the CSP++ framework, their execution speed depends on the way CSP++ implements each feature. Therefore, by writing or structuring CSPm specifications differently, shorter execution times may result from the synthesized system. There are many different structural variations that could have been used to specify the ATM system. In this section we explore the effects of some variations of the ATM with different specification structures. By measuring the performance of the ATM for its different variations, “best practices” for CSPm specifications for CSP++ synthesis can be discovered.

The original ATM modeled each layer (or sub-state machine) of the HCFSM with a separate process so that each process had to communicate with the adjacent layer through channel I/O. Variables were stored using the global variable technique mentioned in section 4.2.2 that uses multiple concurrent processes. Events for which there are no items on the environment stack above a certain point and tracing is not needed are candidates for hiding, to prevent unnecessary stack searching. This was not done in the original ATM model.

To summarize, some of the options for specifying the ATM include the following:

- Hierarchy vs. flat
- Global variables vs. parameters
- No hiding vs. hiding

By mixing and matching some of these options, we try to gain some performance benefit, as shown in Table 9.

Table 9 Time for 10000 ATM transactions (Structure Modifications)

ATM Variations	User Secs.	System Secs.	Total Secs.	% of Max Time
hierarchy, global variables, hiding	17.21	2.6	19.8	100
hierarchy, global variables, no hiding (original)	17.16	2.6	19.76	99.8
flat, global variables, no hiding	11.69	2.26	13.94	70.4
hierarchy, parameters, no hiding	7.62	1.79	9.4	47.5
flat, parameters, no hiding	3.01	0.97	3.98	20.05

Changing the structure of the ATM changes the execution time very significantly. Hiding made practically no difference in execution time. However, flattening the state machine hierarchy cut the execution time of the ATM by about 30%. Using parameters rather than global variables reduced the execution time by over 52%. These two benefits appear to be independent and additive. Ignoring the hiding (since it made no significant difference) and combining the benefits of flattening and parameters, cut the ATM execution time by about 80%. The reduction in execution time is due to the fact that flattening and parameters both reduce the number of threads that are used compared to hierarchy and global variables. The code size is also smaller. Furthermore, a flattened process structure results in much less time spent searching the shallow environment stack. This is something for CSPm specification writers to keep in mind, and points the way to optimizations that the cspt translator could profitably carry out.

5.4 The Effect of CSPm Modifications

The previous work compared the DSS as a CSP++ application with an ObjecTime model of the DSS. In this way, CSP++ could be put into perspective as a code generation tool, by comparing it with another similar tool. ObjecTime generated C++ code from Statecharts and the DSS case study (and later the ATM) used Statecharts to derive CSPm specifications used in C++ code generation via CSP++. Comparing the tools, CSP++ did prove to have a competitive performance. However, in order to ascertain the effect of the modifications to CSP++ since that time, these original tests must be performed again. Since ObjecTime was superceded by Rational Rose RealTime (RRRT), the ObjecTime model of the DSS needed to be recreated in RRRT. Although RRRT uses graphical modeling, can handle real-time constraints, and can generate code for C and Java as well as C++, it does not allow for formal verification as CSP++ does. We used version 6.5.825.0 of RRRT and the ANSI C clock() function to return the elapsed CPU time at the beginning and end of the simulations to make time measurements. The difference between the two times was averaged over 5 runs. In the remainder of this section we use the DSS to compare the new CSP++ with version 3.0 and RRRT and the ATM to compare the new CSP++ with RRRT.

5.4.1 DSS Performance

The times in Table 10 were recorded for the DSS benchmark.

Table 10 Time for 20000 simulated disk accesses

	User Secs.	System Secs.	Total Secs.
RRRT	n/a	n/a	1.1
V3.0 with Pth	29.21	5.49	34.7
V4.1 with Pth	29.89	5.61	35.49

Here we can see that for the DSS case study, CSP++ is about 30 times slower than Rational Rose RealTime. Also, there has been a small percentage increase (2%) in execution time between version 3.0 and 4.1 of CSP++. The ratio of user to system time for the DSS in both versions is about 5.3.

5.4.2 ATM Performance

Just as the DSS benchmark was created by inflating repetitions to a measurable level, so the ATM has received the same treatment. The following shows the modification of the CLIENT repetition for the ATM simulation.

```
SYS = (ATM
          [|{|insertcard,readcard,readpin,choose,getacct,getamnt,
               insertenv,dispense,again,badcard,cancel|}|]
       CLIENT)
          [|{|on,machcash,off|}|]
       OPERATOR \{commit,approved,receipt,startenv}

OPERATOR = on -> machcash!1000000 -> OPERATOR

CLIENT = insertcard -> readcard!1 -> readpin!1 -> choose!3 ->
  getacct!1 -> getamnt!1000000 -> insertenv -> CLIENTCONT(10000)
CLIENTCONT(n) = if (n == 0) then again!0 -> STOP else again!1 ->
  choose!1 -> getacct!1 -> getamnt!1 -> dispense?a -> CLIENTCONT(n-1)
```

The OPERATOR fills the ATM with cash, then the CLIENT performs one deposit followed by 10000 cash withdrawals of $1 each.

For this test, the links to UCFs were removed because the purpose was to measure the performance of the framework, not of the UCF code. We also created an RRRT model of

the same ATM system based on Bjork's Statecharts to provide a similar comparison with RRRT for the ATM as was done for the DSS benchmark.

In Table 11, we compare the performance of the CSP++ and RRRT models of the ATM. In section 5.3 we compared the performance of ATM when specified in variety of different ways in CSPm. RRRT is compared with the fastest and slowest ATM specification variations.

Table 11 Time for 10000 ATM transactions

	User Secs.	System Secs.	Total Secs.
RRRT	n/a	n/a	4.86
V4.1 with Pth (fastest)	6.84	1.61	3.98
V4.1 with Pth (slowest)	17.21	2.6	19.8

For the fastest variation of the ATM case study, CSP++ executed in about 82% of the time it took for Rational Rose RealTime and took just over 4 times the execution time for the slowest ATM variation. This is a lot closer in execution time to RRRT than the DSS case study. The ratio of user to system time for the ATM in V4.1 is about 3.1 for the fastest specification variation and 6.6 for the slowest.

Analysis:

Even with the CSP++'s poor performance with GNU Pth, CSP++ and RRRT have relatively (compared to the DSS) close execution times. If changing the thread model to LinuxThreads makes an order of magnitude difference for the ATM as it did for the DSS, then perhaps changing the thread model would improve the ATM speed. However, the fact that the faster ATM specification version applied the "flat" and "parameters" techniques from section 4.2.1 and section 4.2.2 respectively meant that the amount of

threads used were less than what would have been necessary if the other techniques were applied.

5.5 The Effect of isTop Feature

In section 3.3.4, we discussed introducing the isTop feature to keep CSP++ from searching up the entire tree to look for more EnvSync objects when it was known that no more relevant EnvSync objects existed above. In this section, we explore whether or not this has any effect on the execution time of a system.

To measure the effectiveness of the feature, we wrote a specification intended to make the isTop feature "shine". The specification has many (50000) pairwise synchronizations (between G and H on 'a') at the bottom of a environment stack of depth 3. Dummy synchronizations on 'b' are declared and placed in the environment only to increase the size of the environment stack each Agent needs to search, but no synchronizations on 'b' actually occur. The CSPm is given below without channel declarations.

```
SYS = A [|{b}|] F
A = C [|{b}|] F
C = E [|{b}|] F
E = G(50000) [|{a}|] H(50000)
F = a -> SKIP
G(i) = if i > 0 then a -> G(i-1) else STOP
H(i) = if i > 0 then a -> H(i-1) else STOP
```

The above specification creates an environment stack illustrated by Figure 5, which shows each Agent pointing to its parent in the branching stack.

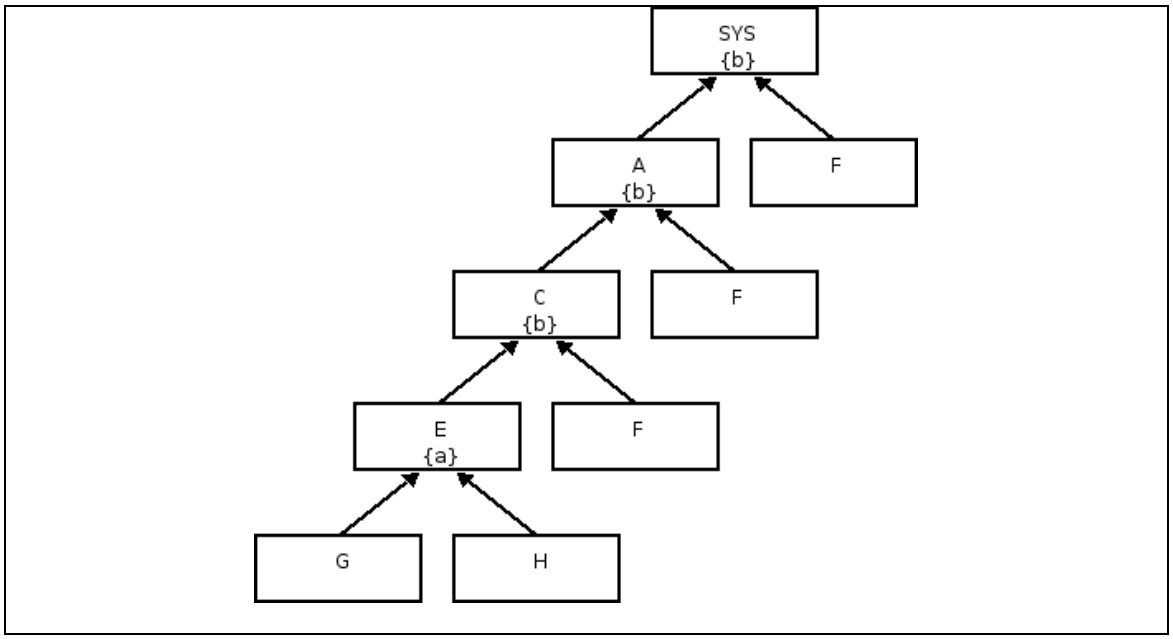


Figure 5 Environment Stack of Specification Demonstrating isTop Feature

Running the program results in 50003 ‘a’ events since each F instance executes ‘a’ once and G and H perform 50000 ‘a’ events together through synchronization. Since the synchronization on ‘a’ only occurs at the bottom, it is wasteful to continue searching the stack in hope of finding higher mentions of ‘a’, and the isTop flag prevents this. We ran the CSP++ program with isTop disabled and enabled and obtained the following results shown in Table 12.

Table 12 Time for 20000 disk accesses (isTop feature)

	User Secs.	System Secs.	Total Secs.
isTop disabled	5.23	0.95	6.18
isTop enabled	5.21	0.95	6.16

As we can easily observe, the isTop feature makes no significant difference in execution time.

Analysis:

The reason isTop may not make much of a difference may be because searching the environment stack seems to already be very efficient and not the bottleneck we imagined.

5.6 Memory Estimates

In Table 13, sizes for the object files the framework, translated case studies (DSS and ATM), and the final executables for version 4.1 of CSP++ can be found. They were obtained using the GNU size utility. The executables marked with "static" include the modules from the C++ and POSIX threads libraries, and those that are not rely on dynamic linking.

Table 13 CSP++ Object File Sizes

Filename / category	Code Sections			
	Text	Data	Bss	Total
Framework files				**28741**
Action.o	8392	268	8	8668
Agent.o	10236	168	1	10405
Lit.o	3716	268	1	3985
task.o	5495	172	16	5683
Translated				
DSS.o	13900	256	912	**15068**
atm.o	28632	572	1724	**30928**
Executable				
DSS (static)	908045	15368	31800	955213
DSS	48286	1680	1940	**51906**
atm (static)	951634	15996	33568	1001198
atm	69174	2088	2944	**74206**

The executable sizes are reasonably small. However, the executables that are linked statically are over 10 times the size of those linked dynamically. This is because CSP++ makes liberal use of the C++ Standard Template Library, iostream, and other bulky packages. So far, no attempt has been made to minimize CSP++'s use of these libraries.

Chapter 6

Conclusions and Future Directions

In this thesis, we presented a reengineered CSP++ to conform with CSPm verification tools. The translator and framework were changed and enhanced to handle a synthesizable subset of CSPm. We demonstrated CSP++ by walking through the selective formalism design flow with a new ATM case study. Performance metrics were taken to position CSP++ with respect to the performance of its previous versions and a competing code generation tool Rational Rose RealTime. In the following sections we will summarize how we achieved our research goals and provide some future direction for continued CSP++ research.

6.1 Conclusions

In the following three sections we explain how reengineering CSP++ to conform with CSPm verification tools enabled us to create a more useful and powerful tool with continued competitive performance.

6.1.1 A More Useful Tool

The csp12 version of CSP++ was limited in its usefulness because of its lack of direct verification tool support. By reengineering CSP++ to conform with commercial CSPm verification tools, software engineers are now able to benefit from a CSP synthesis tool that implements the selective formalism design flow. Whether or not someone is a CSP "guru," they can specify a system in CSPm, verify it to the desired extent with FDR2, explore its behaviour with ProBE, write user-coded functions to link with CSPm events

and interact with the system's environment, synthesize and build a software system. We provided the ATM case study as another example for how to build a system using CSP++, as well as design patterns and tips for better development. These things clearly show that CSP++ has become a more useful tool.

6.1.2 A More Powerful Tool

Although the previous version of CSP++ implemented some complex and powerful features, it was limited in what it could do compared to the CSPm interpretation of CSP. CSP++ can now handle more complex specifications with more elaborate events and the ability to synchronize on the same event over multiple levels. Previous bugs have been fixed through more extensive testing to increase the user's confidence in the tool. User-coded functions have now been fully implemented and tested to allow CSP++ to communicate with its environment. These and other enhancements have made CSP++ a more powerful tool.

6.1.3 Competitive Performance

In the original work on CSP++, it was shown that a CSP++ application performed at a speed competitive with the results of an expensive commercial synthesis tool (Rational Rose RealTime). It was hoped that the reengineering of CSP++ would not adversely affect its performance. Although, CSP++ now performs at an order of magnitude slower than it did before, it is clear what the problem is. We reestablished that the old version of CSP++ was competitive with RRRT performance, found that changing to GNU Pth threads alone caused the noticeable problem, and can now conclude that the underlying thread model needs to be changed to a new model or be returned back to LinuxThreads.

Comparing the old version of CSP++ with Pth with the new CSP++, we found there to be only a small percentage slow down with the new CSP++. We also demonstrated different ways to structure the ATM specification to gain performance benefits. With some minor adjustments to CSP++, it can continue to achieve competitive performance.

6.2 Future Directions

There remain many interesting areas of research within CSP++ and selective formalism. In decreasing order of priority, these include:

- Changing the thread model
- Porting CSP++ to embedded systems
- Enhancing UCF integration
- Implementing more CSPm features
- Supporting more data types
- Finding a way to model time

As well as the above research areas, careful thought must be given as to an efficient way to teach the CSPm skills necessary for development using CSP++. Although not everyone involved in the development needs to know CSP, at least one of the development team should be able to properly write CSP specifications and know how to verify the system's properties. Included in the appendix is a summary of a training seminar for traditional programmers in how to use CSP for CSP++ case study development. We will now discuss the particulars of what work must be done for the above areas of research.

6.2.1 Changing the Thread Model

There are many benefits to using Pth as CSP++'s thread model. Pth is very portable and has kept CSP++ from having to be massaged to work for each different target platform. Since adopting Pth, CSP++ has been ported to x86 Solaris, RedHat, Fedora, and Gentoo. Another benefit of Pth is that it is nonpreemptible, keeping CSP++ from the overhead of frequent context switches and mutexes for protecting critical sections. Furthermore, Pth is freely available for download. However, there are at least a couple drawbacks to using GNU Pth:

1) Slow speed

   One drawback to Pth is that it is slowing down CSP++'s performance enormously. This may not be crucial for some systems where the bulk of execution time is in the UCFs rather than the CSP++ control backbone. However, some systems may require CSP++ to perform at higher speeds.

2) Inappropriate for embedded processors

   As we envision CSP++ to be used in embedded systems, we must consider the platforms we might be using. There has been work underway [Carter, Xu et al. 2005] to port CSP++ to the Xilinx Microblaze processor (a soft processor core for the Virtex FPGA) running uCLinux that, like most operating systems, comes with its own Pthread support. On memory constrained devices, it is not ideal to install an additional redundant Pthread implementation on the system. Furthermore, processors like Microblaze have not had Pth ported to them before—possibly making Pth, and therefore CSP++, difficult to set up on embedded platforms.

Unfortunately, finding an alternative to Pth is not easy. Using a system's default POSIX thread model may run faster than Pth and avoid the headache of porting Pth to an unsupported system, but the preemptibility of such thread models presents a risk. CSP++'s critical sections must be carefully identified and protected (e.g., using mutexes) in the process of moving to preemptible threads, especially in view of the substantial framework changes just made for V4.1. This is despite CSP++ having run on a preemptible thread model before using LinuxThreads in V2.1. In short, Pth causes no immediate threat to CSP++ but, performance and resource concerns do exist.

6.2.2 Porting CSP++ to Embedded Systems

As was mentioned in the previous section, giving CSP++ a role in embedded systems and in hardware/software codesign is a goal of CSP++ research. In order to better understand how CSP++ can be used, we must move CSP++ beyond PCs to other systems that would benefit from CSP++. If CSP++ were ported to an FPGA, the system may be more easily integrated with hardware components that up to now have only been simulated. This would also open a window into new understandings of how UCFs can be used.

In order to do this, we should reduce CSP++'s memory requirements by avoiding the use of bulky C++ runtime libraries. These libraries may also not be supported by cross-compiler and real-time operating system combinations, as we have seen with gcc/uClinux. As well as avoiding C++ runtime libraries, we can eliminate C++ STL use, since STL may not be fully supported either. A minimal system would only need to be able to support the C++ libraries, I/O drivers, CSP++ and its applications.

6.2.3 Enhancing UCF Integration

Although UCFs are working for simple communication, their interface remains fairly primitive. UCFs currently are unable to participate in deterministic choice or in synchronization with other events in the system due to a restriction in [Gardner 2000] stating that events can *either* be used for internal synchronization *or* for linkage to UCFs. The consequences of relaxing this restriction should be investigated to see if CSP++ can be made more flexible in its use of UCFs. Moreover, it is important to more clearly define the rules of usage for UCFs so that we are sure they do not disrupt the verified formal backbone. Ways of validating UCF input should continue to be explored to make UCF integration as seamless as possible.

6.2.4 Implementing More CSPm Features

CSP++ already has a rich set of features for software synthesis. However, there are some that could be added to enhance that feature set, including the '/\' interrupt operator and the '&' boolean guard operator. These operators were discussed in section 3.1.9.

6.2.5 Supporting More Data Types

While integer support is good, it is too simplistic. We have already made some progress in allowing enumerated data types but the processing of the variations of `datatype` definitions needs to be explored first. Supporting more data types would involve significant changes to the translator and framework including their use as process parameters. Of particular interest are the set and sequence types that have the potential to simplify otherwise cumbersome structures (see Queue in section 3.1.9). Since FDR2 does not support floating point numbers or strings, these classic C data types would have to be

modeled differently to work with verification tools. Perhaps CSP++ needs a mechanism for systems to be translated from CSPm specifications without floating point numbers and then permit them in synthesized system. There is no problem with CSP using floating point numbers but Formal Systems has chosen not to implement it in FDR2. The FDR2 manual says that there are workable alternatives for strings using integers and sequences. If CSP++ were used to control a data-dominated system (e.g., digital signal processing), buffers of data could likely be modeled using sequences that can be passed among UCFs that apply filters, calculate transforms, and so on.

6.2.6 Finding a Way to Model Time

Many systems depend on the notion of time with real-time constraints and timeouts. CSP was not designed for time, but extensions and roundabout ways have been introduced since. Schneider talks about the ‘tock’ event in his book [Schneider 2000] that can be used within the existing syntax of CSPm. A ‘tock’ event represents one tick of a clock, and all process definitions that care about timing can synchronize on it. However, lengthy sequences of ‘tock’ events is an awkward way of specifying timing constraints, and would be extremely inefficient to run in CSP++ with many ‘tock’ events synchronizing at tiny intervals. Schneider also discusses Timed CSP, a very clean extension to CSP for time, but there is no commercial verification tool support for it. Perhaps CSP++ could accept Timed CSP operators, but automatically translate the specification to an equivalent ‘tock’-based model for verification purposes.

6.3 Status and Availability of CSP++

Distributions of V4.1 CSP++ for Red Hat 9 and Solaris can be downloaded from W. B. Gardner's website [Bill Gardner / Research Pages] as well as the DSS and ATM case studies. GNU Pth is freely available, as is MySQL for the ATM case study.

References

ARROWSMITH, B. and MCMILLIN, B., 1994. *How to program in CCSP*. Department of Computer Science, University of Missouri-Rolla.

B-core (UK) Ltd. http://www.b-core.com/ [as of 07/25/05].

BETON, R., 2000. libcsp - a Building mechanism for CSP Communication and Synchronisation in Multithreaded C Programs. P.H. WELCH and BAKKERS, ANDR\'E W. P., eds. In: *Communicating Process Architectures 2000*, September 2000, IOS Press. pp. 239-250.

BETON, R., libcsp CSP on Posix Threads. http://sourceforge.net/projects/libcsp/ [as of 07/25/05].

BJORK, R.C., An Example of Object-Oriented Design: An ATM Simulation. http://www.math-cs.gordon.edu/local/courses/cs211/ATMExample/ [as of 06/16/05].

BJORKLUND, D. and LILIUS, J., 2004. Rialto to B: an exercise in formal development of a language for multiple models of computation. *Fourth International Conference on Application of Concurrency to System Design, 2004. ACSD 2004. Proceedings.* 2004, pp. 125-134.

BOUALI, A., Welcome to the Esterel Verification Environment: Xeve. http://www-sop.inria.fr/meije/verification/esterel/ [as of 07/25/05].

BOWEN, J., The B-Method. http://vl.fmnet.info/b/ [as of 07/25/05].

BOWEN, J., The Z notation. http://vl.zuser.org/ [as of 07/25/05].

BROENINK, J.F. and JOVANOVIC, D.S., 2004. Graphical Tool for Designing CSP Systems. EAST, DR. IAN R., P.D. DUCE, D.M. GREEN, MARTIN, JEREMY M. R. and WELCH, PROF. PETER H., eds. In: *Communicating Process Architectures 2004*, 2004, pp. 233-252.

BROENINK, J.F. and JOVANOVIC, D.S., et al, 2002. Controlling a mechatronic setup using Real-time Linux and CTC++. *Proceedings of the 8th Mechatronics Forum International Conference*, 2002, pp. 1323-1331.

BROWN, N.C. and WELCH, PROF. PETER H., 2003. An Introduction to the Kent C++CSP Library. J.F. BROENINK and G.H. HILDERINK, eds. In: *Communicating Process Architectures 2003*, 2003, pp. 139-156.

BUTLER, M.J., 1999. csp2B: A Practical Approach to Combining CSP and B. *FM '99: Proceedings of the Wold Congress on Formal Methods in the Development of Computing Systems-Volume I*, 1999, Springer-Verlag. pp. 490-508.

CARTER, J. and XU, M., et al, 2005. Rapid Prototyping of Embedded Software Using Selective Formalism. *16th IEEE International Workshop on Rapid System Prototyping (RSP'05)*, 2005, pp. 99-104.

CHENG, M.H.M., 1994. *Communicating Sequential Processes: a synopsis*. Department of Computer Science, University of Victoria.

CORBETT, J.C. and DWYER, M.B., et al, 2000. Bandera: extracting finite-state models from Java source code. *ICSE '00: Proceedings of the 22nd international conference on Software engineering*, 2000, ACM Press. pp. 439-448.

DOXSEE, S. and GARDNER, W.B., 2005a. Synthesis of C++ Software from Verifiable CSPm Specifications. *12th IEEE International Conference and Workshops on the Engineering of Computer-Based Systems, 2005. ECBS '05*. 2005a, pp. 193-201.

DOXSEE, S. and GARDNER, W.B., 2005b. Synthesis of C++ software for automated teller from CSPm specifications. *SAC '05: Proceedings of the 2005 ACM symposium on Applied computing*, 2005b, ACM Press. pp. 1565-1566.

Esterel: a Synchronous Reactive Programming Language. http://www-sop.inria.fr/esterel.org/ [as of 06/16/05].

FDR2 User Manual. http://www.fsel.com/documentation/fdr2/html/ [as of 08/18/05].

FERNANDEZ, J. and GARAVEL, H., et al, 1992. A Toolbox For The Verification Of LOTOS Programs. *International Conference on Software Engineering, 1992*. 1992, pp. 246-259.

Formal Systems. http://www.fsel.com/ [as of 07/25/05].

FREITAS, L. and CAVALCANTI, A., et al, 2002. JACK: A Framework for Process Algebra Implementation in Java. *Proceedings of XVI Simpósio Brasileiro de Engenharia de Software*, October 2002, Sociedade Brasileira de Computacao.

GARDNER, W.B., 2000. *CSP++: An Object-Oriented Application Framework for Software Synthesis from CSP Specifications,* Department of Computer Science, University of Victoria, Canada.

GARDNER, W.B., 2003. Bridging CSP and C++ with selective formalism and executable specifications. *First ACM and IEEE International Conference on Formal Methods and Models for Co-Design, 2003. MEMOCODE '03. Proceedings.* 2003, pp. 237-245.

GARDNER, W.B., Bill Gardner / Research Pages. http://www.uoguelph.ca/~gardnerw/research/ [as of 08/18/05].

HILDERINK, G., CSP for Java. http://www.ce.utwente.nl/JavaPP/ [as of 07/25/05].

HINCHEY, M.G. and JARVIS, S.A., 1995. Concurrent Systems: Formal Development in CSP. 1995, McGraw-Hill International.

HINCHEY, M.G. and RASH, J.L., et al, 2005. A Formal Approach to Requirements-Based Programming. *12th IEEE International Conference and Workshops on the Engineering of Computer-Based Systems, 2005. ECBS '05.* 2005, pp. 339-345.

HOARE, C. A. R., 1985. Communicating Sequential Processes. J. DAVIES, ed. In: 1985, Prentice Hall International. pp. 100-106.

JACKSON, P.A. and HUTCHINGS, B.L., et al, 2003. Simulation and Synthesis of CSP-based Interprocess Communication. *11th Annual IEEE Symposium on Field-Programmable Custom Computing Machines,* 2003, .

MCEWAN, A.A., 2004. A Calculated Implementation of a Control System. EAST, DR. IAN R., P.D. DUCE, D.M. GREEN, MARTIN, JEREMY M. R. and WELCH, PROF. PETER H., eds. In: *Communicating Process Architectures 2004,* 2004, pp. 265-280.

MOORES, J., 1999. CCSP - A Portable CSP-Based Run-Time System Supporting C and occam. B.M. COOK, ed. In: *Proceedings of WoTUG-22: Architectures, Languages and Techniques for Concurrent Systems,* 1999, pp. 147-169.

On-The-Fly, LTL Model Checking with SPIN. http://spinroot.com/ [as of 07/25/05].

PHILLIPS, J.D. and STILES, G.S., 2004. An Automatic Translation of CSP to Handel-C. EAST, DR. IAN R., P.D. DUCE, D.M. GREEN, MARTIN, JEREMY M. R. and WELCH, PROF. PETER H., eds. In: *Communicating Process Architectures 2004,* 2004, pp. 19-38.

RAJU, V. and RONG, L., et al, 2003. Automatic Conversion of CSP to CTJ, JCSP, and CCSP. J.F. BROENINK and G.H. HILDERINK, eds. In: *Communicating Process Architectures 2003*, 2003, pp. 63-81.

Rebeca Home Page. khorshid.ece.ut.ac.ir/~rebeca/ [as of 07/25/05].

ROSCOE, A.W., 1998. The Theory and Practive of Concurrency. HOARE, C. A. R. and R. BIRD, eds. In: 1998, Prentice Hall International.

SCHNEIDER, S., 2000. Concurrent and Real-time Systems: The CSP Approach. 2000, John Wiley & Sons, Ltd.

SCHNEIDER, S., Concurrent and Real-time Systems: the CSP Approach. http://www.cs.rhbnc.ac.uk/books/concurrency/ [as of 08/18/05].

SIRJANI, M. and SHALI, A., et al, 2004. A front-end tool for automated abstraction and modular verification of actor-based models. *Fourth International Conference on Application of Concurrency to System Design, 2004. ACSD 2004. Proceedings.* 2004, pp. 145-148.

SOMMERVILLE, I., 2000. Software Engineering. 2000, Addison Wesley.

STEPNEY, S., 2003. *CSP / FDR2 to Handel-C translation.* University of York.

VAHID, F. and GIVARGIS, T., 2002. Embedded System Design: A Unified Hardware/Software Introduction. 2002, John Wiley & Sons.

WELCH, P.H. and MARTIN, J.M.R., 2000. A CSP model for Java multithreading. *International Symposium on Software Engineering for Parallel and Distributed Systems, 2000. Proceedings.* 2000, pp. 114-122.

World-wide Environment for Learning LOTOS (WELL) - Introduction. http://www.cs.stir.ac.uk/~kjt/research/well/ [as of 07/25/05].

Appendix A

ATM Statecharts

This appendix provides the Statecharts from Bjork's ATM requirements documents [An Example of Object-Oriented Design: An ATM Simulation]. They form a hierarchical concurrent finite state machine (HCFSM). The overall ATM is the top level Statechart, shown in Figure 6. After being turned on, it alternates between the IDLE state and SERVING CUSTOMER. The SERVING CUSTOMER state represents a single Session, which is described by the next level Statechart (Figure 7).

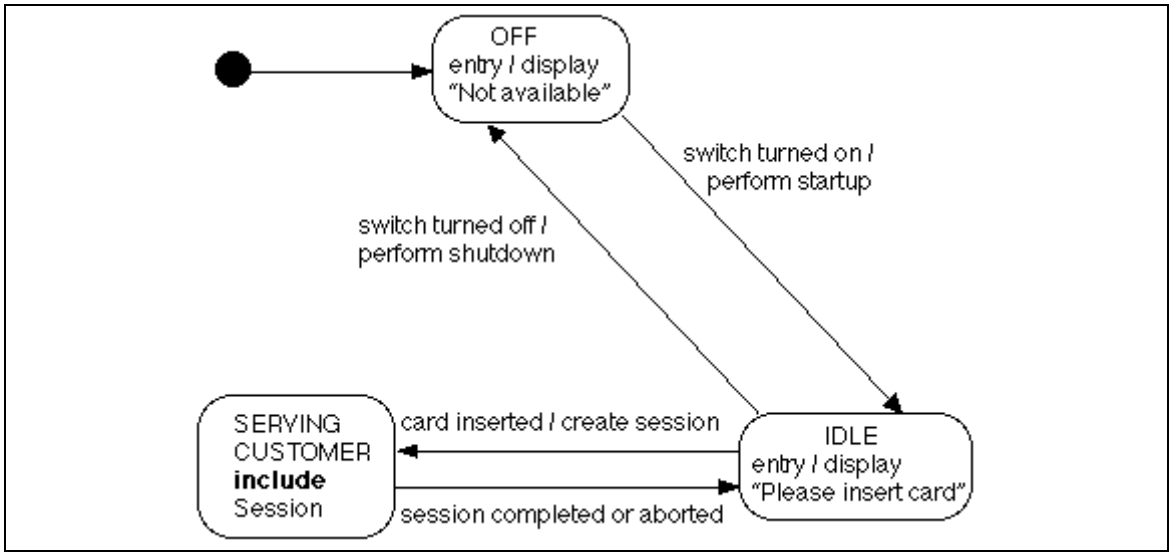


Figure 6 Statechart for Overall ATM

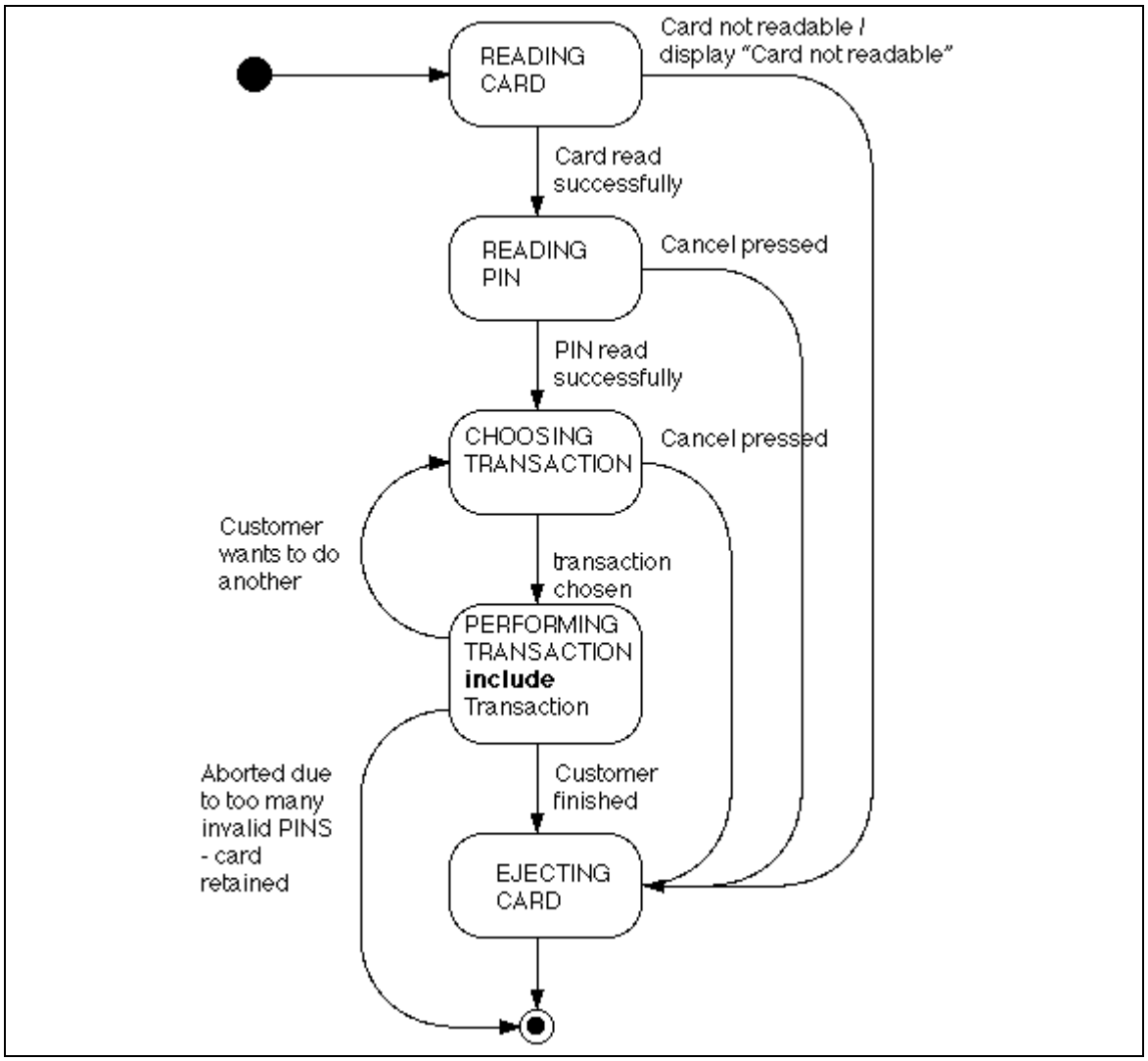


Figure 7 Statechart for One Session

The PERFORMING TRANSACTION in the Session Statechart above is where the next level Statechart shown in Figure 8, Transaction, is included.

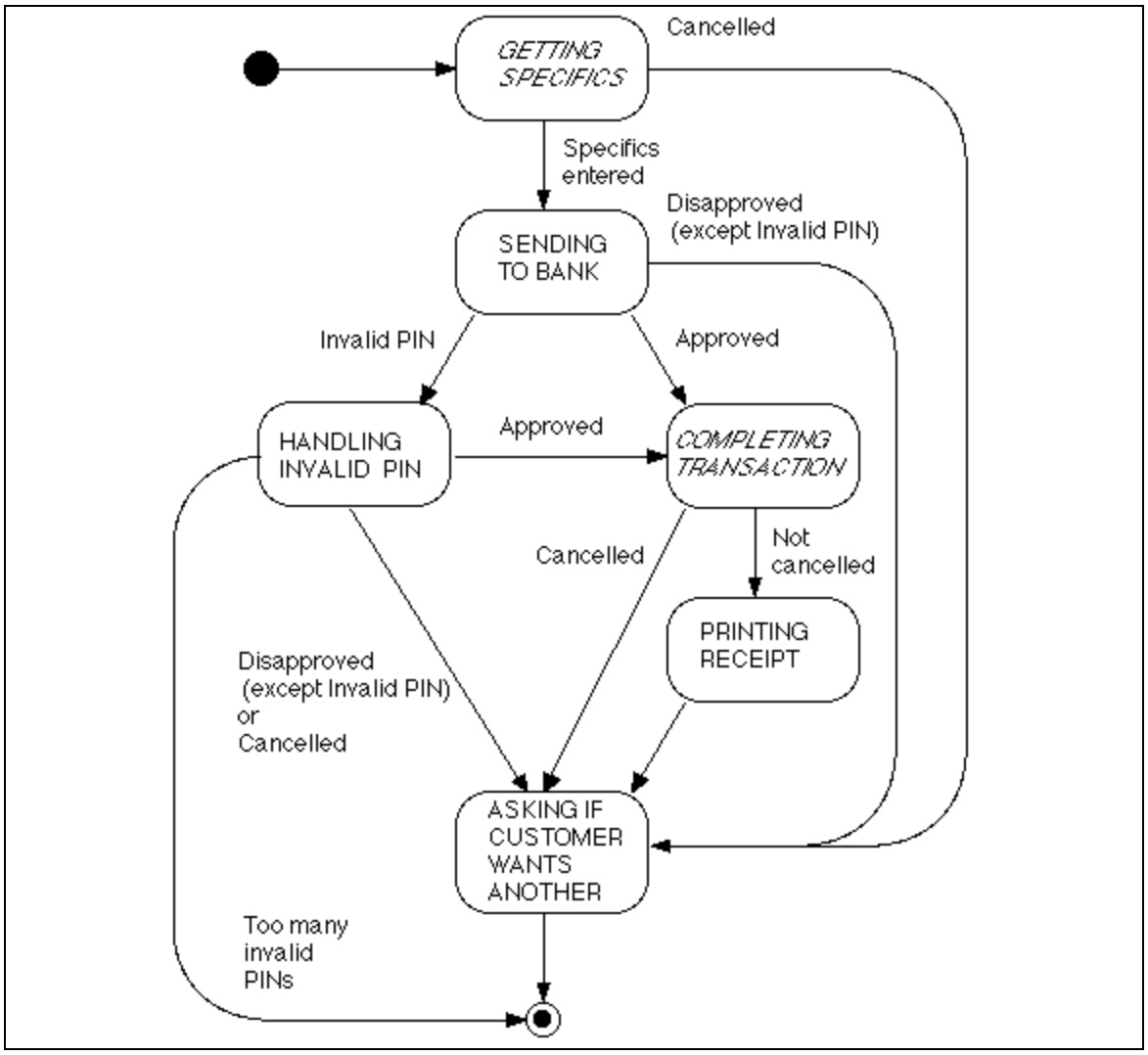


Figure 8 Statechart for One Transaction

The italicized operations in the above Transaction Statechart indicate that there is unique unspecified behaviour for each transaction type (i.e., withdrawal, deposit, transfer, account balance). These unspecified behaviours were defined in the ATM case study in the implementation model.

Appendix B

ATM Code

This appendix contains the CSPm and UCF code for the ATM case study (the Bank simulator code is not included). The code for this ATM system comes from 3 main sources: atm.csp, atmprocs.cc, and bank.cc.

- **atm.csp** is where the backbone of the ATM is specified. It is translated by cspt into atm.cc that is linked with user-coded functions in atmprocs.cc.
- **atmprocs.cc** is how the ATM interfaces with its environment. It allows user input, displays output, and communicates through UDP sockets with bank.cc.
- **bank.cc** is not part of the ATM but is an important part of the overall system. The ATM sends requests to the simulated bank that are handled by translating as MySQL database queries.

Setting up the System

The atmprocs.cc and bank.cc files may need to be adjusted for the target system's configuration with regards to IP addresses, MySQL users/password/databases/etc. The following "root"-user MySQL table "CA" (short for "client account") in the "mydb" database is assumed to be intact when the system is started.

```
+----------+---------+------+-----+---------+-------+
| Field    | Type    | Null | Key | Default | Extra |
+----------+---------+------+-----+---------+-------+
| ClientID | int(11) |      | PRI | 0       |       |
| Card     | int(11) |      |     | 0       |       |
| PIN      | int(11) |      |     | 0       |       |
| Checking | int(11) |      |     | 0       |       |
| Savings  | int(11) |      |     | 0       |       |
+----------+---------+------+-----+---------+-------+
```

The above database schema can be created using the following commands.

- `create dttabase mydb;`
- `use mydb;`
- `create table CA( ClientID int(11) not null primary key,Card int(11) not null, PIN int(11) not null, Checking int(11) not null, Savings int(11) not null);`

The database can be populated with the following command (gives the person with Client #1 with Card #1 and PIN #1, 5 dollars in their Checking and Savings accounts):

> insert into CA(ClientID,Card,PIN,Checking,Savings) values(1,1,1,5,5);

The atm program must be built (recommend Makefile) and the bank program must be created using the command at the top of the bank.cc file.

Running the System

- Open two terminal windows
- Run 'bank' in one
- Run 'atm -t -i' in the other
- Follow the instructions of the ATM system and press CTRL-C to exit each program

In the following sections the ATM CSPm, UCFs, and Bank will be presented.

B.1 CSPm Specification for ATM

```
--atm.csp: Created by Stephen Doxsee, March 9, 2005

--//////////////
--Channels
--//////////////

channel anothertrans, approved, badcard, cancel, commit, ejectcard,
endtrans, exceedsMch, holdingcard, insertcard, insertcard_i, insertenv,
invalidPIN, notapproved, notcancel
channel off, on, receipt, rollback, sameacct, sessiondone, startenv
channel banksend, checkdata:
choices.integers.integers.integers.integers.integers.integers

channel bankstatus, reportdata: integers.integers.integers.integers

channel again, amntget, amntset, balget, balset, cardget, cardset,
dispense, display, getacct, getamnt, getfrom, getto, machcash, machget,
machset, pinget, pinset, readcard, readnewpin, readpin, request :
integers

channel choose, choose_i: choices

--//////////////
--Types
--//////////////

nametype integers = {0..2}

nametype choices = {1..4}

--//////////////
--We use variables in this way to avoid passing large amounts of
information as parameters to processes
--//////////////

VARIABLES = AMNTi ||| BALi ||| CARDi ||| MACHi ||| PINi

AMNTi = amntset?x -> AMNT(x)
AMNT(val) = amntset?x -> AMNT(x)
   [] amntget!val -> AMNT(val)

BALi = balset?x -> BAL(x)
BAL(val) = balset?x -> BAL(x)
        [] balget!val -> BAL(val)

CARDi = cardset?x -> CARD(x)
CARD(val) = cardset?x -> CARD(x)
        [] cardget!val -> CARD(val)

MACHi = machset?x -> MACH(x)
MACH(val) = machset?x -> MACH(x)
        [] machget!val -> MACH(val)
```

```
PINi = pinset?x -> PIN(x)
PIN(val) = pinset?x -> PIN(x)
   [] pinget!val -> PIN(val)

--//////////////////
-- OVERALL: State machine that models the overall system behaviour
--//////////////////

OVERALL = OFF

OFF = on -> machcash?cash -> machset!cash -> IDLE

-- insertcard synchronizes with session (insertc is a workaround to
allow a third process to synchronize)

IDLE = insertcard -> insertcard_i -> SERVING
   [] off -> OFF

SERVING = sessiondone -> IDLE

--//////////////////
-- SESSION: State machine that models a session
--//////////////////

SESSION = insertcard_i -> READINGCARD

READINGCARD = readcard?c -> (cardset!c -> READINGPIN)
   [] badcard -> EJECT

READINGPIN = readpin?p -> (pinset!p -> CHOOSING)
   [] cancel -> EJECT

CHOOSING = choose?menu -> (choose_i!menu -> TRANS)
   [] cancel -> EJECT

TRANS = endtrans -> EJECT
   [] anothertrans -> CHOOSING
   [] holdingcard -> DONE

EJECT = ejectcard -> DONE

DONE = sessiondone -> SESSION

--//////////////////
-- TRANSACTION: State machine that models a transaction
--//////////////////

TRANSACTION = choose_i?menu -> SPECIFICS(menu)

--// WITHDRAWL
SPECIFICS(1) =
   (getacct?account ->
   (getamnt?amount -> (amntset!amount -> SEND(1,account,1,1,amount) )
   [] cancel -> ANOTHER)
   [] cancel -> ANOTHER) -- withdraw

--// TRANSFER
```

```
SPECIFICS(2) =
   (getfrom?f ->
   (getto?t ->
   if (f == t) then sameacct -> SPECIFICS(2)
   else (getamnt?a -> (amntset!a -> SEND(2,1,f,t,a) )
   [] cancel -> ANOTHER)
   [] cancel -> ANOTHER)
   [] cancel -> ANOTHER)  -- transfer

-- provide timeout for envelope!!!!
--// DEPOSITS
SPECIFICS(3) =
   (getacct?account ->
   (getamnt?amount -> (amntset!amount -> SEND(3,account,1,1,amount) )
   [] cancel -> ANOTHER)
   [] cancel -> ANOTHER) -- deposit

--// ACCOUNT BALANCE
SPECIFICS(4) =
   (getacct?account -> SEND(4,account,1,1,0)
   [] cancel -> ANOTHER) -- info

SEND(m,account,from,to,amount) = cardget?c -> pinget?p ->
banksend!m.c.p.account.from.to.amount -> RECEIVE(m)

--// get response from bank on what was requested
-- try sending information to bank to see if transaction is ok
-- bankstatus (WITHDRAW)
--   approvedstat == 0 means not approved
--   pinstat == 0 means invalid PIN
RECEIVE(menu) = bankstatus?approvedstat.pinstat.pin.val ->
   (if (pinstat == 0) then invalidPIN ->
HANDLEPIN(menu,approvedstat,pin,val,1)
   else if (approvedstat != 0) then approved -> COMPLETING(menu,val)
   else rollback -> ANOTHER)

--// 3 unsuccessful pin entries -> hold the card in machine
HANDLEPIN(m,approvedstat,realpin,val,attempt) =
   if( attempt >= 3 ) then
      rollback -> holdingcard -> TRANSACTION
   else
      readnewpin?p -> pinset!p ->
      if( p == realpin ) then (if( approvedstat != 0 ) then approved ->
COMPLETING(m,val) else rollback -> ANOTHER)
      else invalidPIN -> HANDLEPIN(m,approvedstat,realpin,val,attempt+1)

--// check to see if the machine has the money required
COMPLETING(1,balance) = machget?m -> amntget?amount ->
   if (m >= amount) then
      commit -> machset!m-amount -> dispense!amount -> RECEIPT
   else rollback -> exceedsMch -> ANOTHER

COMPLETING(2,v) = commit -> RECEIPT

COMPLETING(3,v) = startenv ->
   (insertenv -> (commit -> RECEIPT)
   [] cancel -> (rollback -> ANOTHER))
```

```
COMPLETING(4,balance) = commit -> display!balance -> RECEIPT

RECEIPT = receipt -> ANOTHER

--// more transactions or exit
ANOTHER = again?x ->
       if( x == 1 ) then anothertrans -> TRANSACTION
       else endtrans -> TRANSACTION

--//////////////////
-- ATM: This models the ATM
--//////////////////

ATM = ((OVERALL
   [|{|insertcard_i,sessiondone|}|] SESSION )
   [|{|choose_i,endtrans,anothertrans,holdingcard|}|] TRANSACTION)
   [|{|cardset,cardget,pinset,pinget,machset,machget,balset,balget,amnts
et,amntget|}|] VARIABLES


--//////////////////
-- SYS: This models the entire system
--//////////////////

--SYS = ((ATM
-- [|{|banksend,bankstatus, commit|}|] BANK)
-- [|{|insertcard,readcard,readpin,choose,getacct,getamnt,dispense,
     again,badcard,cancel|}|] CLIENT )
-- [|{|on,machcash,off|}|] OPERATOR

SYS = ATM [|{|on,off|}|] OPERATOR

--//////////////////
-- BANK: This models the bank's interface to the ATM
-- bankstatus is status (badpin,approved,other), pin, value
-- (return value of some kind)
--//////////////////

BANK = banksend?inq.c.p.account.t.f.amount ->
checkdata!inq.c.p.account.t.f.amount -> reportdata?w.x.y.z ->
bankstatus!w.x.y.z ->
   if( inq == 3) then commit -> BANK
   else BANK

--OPERATOR = on -> machcash!2 -> off -> OPERATOR
OPERATOR = on -> off -> OPERATOR

--//////////////////
-- CLIENT: This models a client using the ATM
--//////////////////

CLIENT = insertcard -> readcard!1 -> readpin!1 -> choose!1 -> getacct!1
-> getamnt!2 -> dispense?a -> again!0 -> SKIP

--//////////////////
-- This is the formal verification section
```

```
--/////////////////


-- if you get 3 invalidPIN messages without starting a new transaction,
your card should be held
assert ATM \ diff(Events,{|invalidPIN, again, receipt, holdingcard|})
[T= invalidPIN -> invalidPIN -> invalidPIN -> holdingcard -> STOP

-- should fail since a receipt can't be issued after 3 invalid pins in
one session
assert ATM \ diff(Events,{|invalidPIN, again, receipt, holdingcard|})
[T= invalidPIN -> invalidPIN -> invalidPIN -> receipt -> STOP

P = on -> machcash?x -> off -> P
assert P [F= ATM \ diff(Events,{|on, off,machcash|})

assert ATM \ diff(Events,{|readcard, readpin, insertcard|}) [T=
insertcard -> readcard?x -> readpin?x -> STOP

assert OVERALL :[deadlock free [F]]
assert OVERALL :[livelock free [F]]
assert OVERALL :[deterministic [F]]
assert SESSION :[deadlock free [F]]
assert SESSION :[livelock free [F]]
assert SESSION :[deterministic [F]]
assert TRANSACTION :[deadlock free [F]]
assert TRANSACTION :[livelock free [F]]
assert TRANSACTION :[deterministic [F]]
assert ATM :[deadlock free [F]]
assert ATM :[livelock free [F]]
assert ATM :[deterministic [F]]
```

B.2 User-Coded Functions

This section contains all the user-coded functions for the ATM. They are provided in one file ‘atmprocs.cc’. Below is a table showing all the UCFs in the ATM case study with their UCF names, purpose, and invocation from CSPm.

Table 14 UCFs for the ATM

UCF name	Purpose	CSPm invocation
machcash_chanInput	inputs the amount of cash from the operator	machcash?x
insertcard_atomic	simulates the insertion of a card	insertcard
readcard_chanInput	read the card number	readcard?c

Table 14 UCFs for the ATM

UCF name	Purpose	CSPm invocation
readpin_chanInput	read the pin number	readcard?p
choose_chanInput	choose the type of transaction to do	choose?menu
getacct_chanInput	choose between the savings and checking account	getacct?account
getamnt_chanInput	input the amount of money to be withdrawn/transferred/deposited	getamnt?amount
getfrom_chanInput	choose the account to transfer the money from	getfrom?f
getto_chanInput	choose the account to transfer the money to	getto?t
banksend_chanOutput	send the transaction request to the bank	banksend!m.c.p.account. from.to.amount
bankstatus_chanInput	input the return status of the transaction from the bank	bankstatus?approvedstat. pinstat.pin.val
commit_atomic	tell the bank that the transaction is complete	commit
rollback_atomic	tell the bank to undo the uncompleted transaction	rollback
again_chanInput	choose whether or not to do another transaction	again?x
startenv_atomic	start the envelope deposit slot	startenv
insertenv_atomic	insert the envelope in the slot	insertenv
exceedsMch_atomic	notify the client that there is not enough money in the machine for the requested withdrawal	exceedsMch
dispense_chanOutput	dispense the requested amount of cash	dispense!amount
readnewpin_chanInput	try reading the pin again if the previous one was invalid	readnewpin?p
sameacct_atomic	notify the client that they are transferring to the same account	sameacct
display_chanOutput	display to the client the balance of the account	display!amount

Table 14 UCFs for the ATM

UCF name	Purpose	CSPm invocation
cancel_atomic	the client presses the cancel button	cancel

The UCF code is presented below.

```
/*
 * atmprocs.cc
 *
 * External action procedures
 */

#include "Lit.h"
#include "List.h"  // STL wrapper
#include "Action.h"
#include <string>
#include <map>

#include <mysql.h>
#include <stdio.h>
#include <stdlib.h>
#include <errno.h>
#include <string.h>
#include <strings.h>
#include <sys/types.h>
#include <netinet/in.h>
#include <netdb.h>
#include <sys/socket.h>
#include <sys/wait.h>

#define THEIRPORT 4950     /* the port we will be sending to */
#define MYPORT 4951     /* the port we will be receiving from */

#define MAXBUFLEN 100

int sockfd;
struct sockaddr_in their_addr;  /* their address information */
struct hostent *he;
struct sockaddr_in my_addr;  /* our address information */
socklen_t addr_len;
int numbytes;
char buf[MAXBUFLEN];
char thehost[] = "127.0.0.1";

using namespace std;

int transaction, card, pin, account, from, to, amount;

void machcash_chanInput( ActionType t, ActionRef* a, Var* v, Lit* l )
{
   cout << "--mashcash--" << endl;
   int machcash;
   cout << "Operator, how much cash will the machine hold?" << endl;
```

```
   cout << "Please enter the amount -> ";
   cin >> machcash;
   *v = Lit(machcash);  // this is the input val
}

void insertcard_atomic( ActionType t, ActionRef* a, Var* v, Lit* l )
{
   cout << "--insertcard--" << endl;
   cout << "...listening and waiting for card to be inserted" << endl;
   /*
    * put sensor wait here
    */
   cout << "Card Inserted" << endl;
}

void readcard_chanInput( ActionType t, ActionRef* a, Var* v, Lit* l )
{
   cout << "--readcard--" << endl;
   int cardnumber;
   cout << "Please enter your Card number -> ";
   cin >> cardnumber;
   *v = Lit(cardnumber);// this is the input val
}

void readpin_chanInput( ActionType t, ActionRef* a, Var* v, Lit* l )
{
   cout << "--readpin--" << endl;
   int pinnumber;
   cout << "Welcome to the CSP++ ATM" << endl;
   cout << "Please enter your PIN -> ";
   cin >> pinnumber;
   *v = Lit(pinnumber);  // this is the input val
}

void choose_chanInput( ActionType t, ActionRef* a, Var* v, Lit* l )
{
   cout << "--choose--" << endl;
   int menu;
   cout << "What type of transaction would you like to do?" << endl;
   cout << "\n1) Cash Withdrawal\n2) Transfer\n3) Deposit\n4) Account Balance\n"
<< endl;
   cout << "Please enter your choice -> ";
   cin >> menu;
   *v = Lit(menu); // this is the input val
}

void getacct_chanInput( ActionType t, ActionRef* a, Var* v, Lit* l )
{
   cout << "--getacct--" << endl;
   int account;
   cout << "Which account would you like to use?" << endl;
   cout << "\n1) Checking\n2) Savings\n\n" << endl;
   cout << "Please enter your choice -> ";
   cin >> account;
   *v = Lit(account); // this is the input val
}

void getamnt_chanInput( ActionType t, ActionRef* a, Var* v, Lit* l )
{
   cout << "--getamnt--" << endl;
   int amount;
   cout << "Please enter the amount -> ";
   cin >> amount;
```

```
   /*
    * note: this machine will allow any amount (not just multiples of $20
    * for withdrawals)
    */
   *v = Lit(amount);  // this is the input val
}

void getfrom_chanInput( ActionType t, ActionRef* a, Var* v, Lit* l )
{
   cout << "--getfrom--" << endl;
   int from;
   cout << "Which account would you like to transfer from?" << endl;
   cout << "\n1) Checking\n2) Savings\n\n" << endl;
   cout << "Please enter your choice -> ";
   cin >> from;
   *v = Lit(from); // this is the input val
}

void getto_chanInput( ActionType t, ActionRef* a, Var* v, Lit* l )
{
   cout << "--getto--" << endl;
   int to;
   cout << "Which account would you like to transfer to?" << endl;
   cout << "\n1) Checking\n2) Savings\n\n" << endl;
   cout << "Please enter your choice -> ";
   cin >> to;
   *v = Lit(to);// this is the input val
}

void banksend_chanOutput( ActionType t, ActionRef* a, Var* v, Lit* l )
{
   if ((he=gethostbyname(thehost)) == NULL) { /* get the host info */
      herror("gethostbyname");
      exit(1);
   }

   if ((sockfd = socket(AF_INET, SOCK_DGRAM, 0)) == -1) {
      perror("socket");
      exit(1);
   }

   their_addr.sin_family = AF_INET;        /* host byte order */
   their_addr.sin_port = htons(THEIRPORT); /* short, network byte order */
   their_addr.sin_addr = *((struct in_addr *)he->h_addr);
   bzero(&(their_addr.sin_zero), 8);     /* zero the rest of the struct */

   List<Lit>* temp = l->getList();
   transaction = int((*temp)[0]);
   card = int((*temp)[1]);
   pin = int((*temp)[2]);
   account = int((*temp)[3]);
   from = int((*temp)[4]);
   to = int((*temp)[5]);
   amount = int((*temp)[6]);

   if( transaction == 2) // transfer
      account = from;

   sprintf(buf,"process,%d,%d,%d,%d,%d,%d,%d",
        transaction,card,pin,account,from,to,amount);

   if ((numbytes=sendto(sockfd, buf, strlen(buf), 0,
        (struct sockaddr *)&their_addr, sizeof(struct sockaddr))) == -1)
```

```
   {
      perror("sendto");
      exit(1);
   }
}

void bankstatus_chanInput( ActionType t, ActionRef* a, Var* v, Lit* l )
{
   addr_len = sizeof(struct sockaddr);
   if ((numbytes=recvfrom(sockfd, buf, MAXBUFLEN, 0,  \
        (struct sockaddr *)&their_addr, &addr_len)) == -1)
   {
      perror("recvfrom");
      exit(1);
   }
   buf[numbytes] = '\0';

   // Create a datum to return to the CSP backbone
   DatumID tempID = "bankstatus_d";
   char *ptr;
   ptr = strtok(buf,",");
   Lit a1 = Lit(atoi(ptr));
   ptr = strtok(NULL,",");
   Lit a2 = Lit(atoi(ptr));
   ptr = strtok(NULL,",");
   Lit a3 = Lit(atoi(ptr));
   ptr = strtok(NULL,",");
   Lit a4 = Lit(atoi(ptr));

   // this will be the value given to the channel input
   *v = Lit(tempID, new List<Lit>(a1,a2,a3,a4) );
}

void commit_atomic( ActionType t, ActionRef* a, Var* v, Lit* l )
{
   sprintf(buf,"commit");

   if ((numbytes=sendto(sockfd, buf, strlen(buf), 0, \
         (struct sockaddr *)&their_addr, sizeof(struct sockaddr))) == -1)
   {
      perror("sendto");
      exit(1);
   }
}

void rollback_atomic( ActionType t, ActionRef* a, Var* v, Lit* l )
{
   sprintf(buf,"rollback");

   if ((numbytes=sendto(sockfd, buf, strlen(buf), 0,
         (struct sockaddr *)&their_addr, sizeof(struct sockaddr))) == -1)
   {
      perror("sendto");
      exit(1);
   }
}

void again_chanInput( ActionType t, ActionRef* a, Var* v, Lit* l )
{
   cout << "--again--" << endl;
   int yesorno;
   cout << "Would you like to make another transaction (0 - NO, 1 - YES) -> ";
   cin >> yesorno;
```

```
   *v = Lit(yesorno); // this is the input val
}

void startenv_atomic( ActionType t, ActionRef* a, Var* v, Lit* l )
{
   cout << "--startenv--" << endl;
   cout << "Please insert your envelope" << endl;
}

void insertenv_atomic( ActionType t, ActionRef* a, Var* v, Lit* l )
{
   cout << "--insertenv--" << endl;
   cout << "Envelope inserted. Thankyou." << endl;
}

void exceedsMch_atomic( ActionType t, ActionRef* a, Var* v, Lit* l )
{
   cout << "--exceedsMch--" << endl;
   cout << "Sorry, there is not enough money in the machine to meet your
withdrawal request." << endl;
}

void dispense_chanOutput( ActionType t, ActionRef* a, Var* v, Lit* l )
{
   cout << "--dispense--" << endl;
   cout << "Dispensing $" << *l << endl;
}

void readnewpin_chanInput( ActionType t, ActionRef* a, Var* v, Lit* l )
{
   cout << "--readnewpin--" << endl;
   int pinnumber;
   cout << "Try entering the correct PIN -> ";
   cin >> pinnumber;
   *v = Lit(pinnumber); // this is the input val
}

void sameacct_atomic( ActionType t, ActionRef* a, Var* v, Lit* l )
{
   cout << "--sameacct--" << endl;
   cout << "Error, you are transfering to the same account. Please choose your
accounts again." << endl;
}

void display_chanOutput( ActionType t, ActionRef* a, Var* v, Lit* l )
{
   cout << "--display--" << endl;
   cout << "The balance in your " << (account == 1 ? "Checking" : "Savings") <<
" account is $" << *l << endl;
}

void cancel_atomic( ActionType t, ActionRef* a, Var* v, Lit* l )
{
   cout << "--cancel--" << endl;
   cout << "Will you cancel?" << endl;
}
```

B.3 Bank Simulation

The bank simulation runs along side atm program generated from CSP++. It listens for and responds to messages from the atm user-coded functions and makes MySQL database connections to the bank database holding account information. Depending on the setup of the target system (below is Fedora Core 3), bank.cc can be compiled as follows:

```
> g++ bank.cc -o bank -I/usr/local/mysql/include -L/usr/local/mysql/lib/ -lmysqlclient -lz
```

The MySQL database information that is listed above the main() function in the code may need to be changed as well as the `thehost` variable listed at the beginning of main().

Appendix C

Training CSP++ Personnel

We are interested in promoting the selective formalism design flow approach via CSP++ to industry, but realize that it is unrealistic to subject software engineers to months of training in formal methods. Therefore, we developed a minimal set of training intended to equip someone to work with CSP++ and the Formal Systems tools. To this end, we practiced on two computer science graduate students who were new to formal methods and to CSP by having them take the CSP training.

In this appendix, we identify and describe three roles in the selective formalism design flow that should be filled in order to best utilize CSP++:

1) CSPm specifier
2) UCF coder
3) Verification “guru”

Each of these roles are important but require different kinds of training. We will now look at what training is necessary for competence in these roles.

1) CSPm Specifier

In order to train people for CSP++ application development in a brief amount of time, we designed a CSP seminar course of six two-hour sessions with the objective of preparing students with the knowledge and ability to effectively and efficiently design and develop software systems from the formal process algebra, CSP. It was not our intention to provide students with an exhaustive knowledge of CSP and formal methods but rather to

provide them with an understanding of how and when CSP++ could be applied to software engineering. The course was a study of the formal process algebra, CSP, its use in modeling computer systems, and the use of simulation, verification, and software synthesis tools for it.

We worked through textbook examples [Schneider 2000], tutorials or practicals [Concurrent and Real-time Systems: the CSP Approach], and will use tools like Probe and FDR2. We also used the CSP++ framework to synthesize software and linked UCFs into the CSP++ backbone. In order to provide a controlled training environment where the only learning was in the classroom, no homework was given to the students. This provided us with a better idea of how long it takes to train personnel for CSP++ use. The following table shows the course outline.

Table 15 CSP Seminar Outline

DAY 1	DAY 2	DAY 3
Sequential Processes * Events and Processes * Performing Events * Recursion * Choice Concurrency * Alphabetized * Interleaving * Interface	Abstraction and Control Flow * Hiding * Renaming * Sequential Composition * Interrupt	Traces Refinement Failures

DAY 4	DAY 5	DAY 6
Tools * Checker * Probe * FDR2 Learn to debug	CSP++ * Demo * "Open the framework hood" Design Flow * The 4 models * Statecharts * Design Patterns	Simple Case Studies showing design flow Write own programs Use user coded functions

The seminar was taken by two computer science graduate students who had no experience with formal methods or verification but had a substantial knowledge of C++. Based on observation during the seminar and comments from the students at the end of the seminar, here are some highlights of the things that went well and the things that could be improved.

Things that went well

We were able to cover a lot of material in a short amount of time. Many of the topics were well understood and seen to be useful for the purpose of the course. In particular, the material from the first two and last two days were appreciated. The students felt prepared to implement a small CSP++ system on their own as they understood enough of the language, the tools, and the selective formalism design flow. In fact, soon afterwards, both worked on a "Point of Sale" case study that resulted in a conference paper [Carter, Xu et al. 2005].

Things that could be improved

Students found that the material from the middle sessions on verification was difficult to understand and not particularly useful. This may have been because the examples were too simple to be applied to real specifications later on. They were unable to do more than the automatic verification provided by FDR2. It was also difficult for the students to remember the material from week to week without homework.

The seminar succeeded in training personnel in the basics of developing CSP++ applications and preparing them to fit the role of "UCF coder" or "CSPm specifier" for the selective formalism design flow. With an understanding of the basic CSPm elements and purpose of the four complementary models (see section 4.1) in the selective formalism design flow, a person could become a CSPm specifier.

2) UCF coder

Typically, a UCF coder would not need to know a lot about CSP. Obviously, someone would need to understand programming and software engineering. Furthermore, as CSP++ is eventually intended for embedded systems, those fulfilling this role may also benefit from training in lower level programming that would provide them with the background to connect CSP++ with hardware components. A suitable UCF coder would be a graduate of a computer science bachelors program but may also include computer engineers or people with similar training. In theory, UCF coders would only need to be provided with the CSP++ API for UCFs and requirements for the function's purpose. The current CSP++ API for UCFs only has a single form of function prototype but could be expanded or enhanced in the future. Someone with an understanding of CSP++ should

bring UCF coders up-to-speed with the rules and principles of UCF use. Some of these principles were discussed in section 4.4.

3) Verification “Guru”

The portion of the CSP seminar described above that focused on verification did not prepare students to fit the role of “verification guru”. Someone in this role would need to have had detailed training in formal methods, the CSP language, writing verification statements, and “asking the right questions” of the specification. This person may not need to know much about C++ but could come from a university program with a more theoretical mathematics emphasis.