Extending the CSP++
Object Oriented Application Framework

Daniel Garner

September 2012

Abstract
This project extends the subset of CSP\textsubscript{M} that can be translated using the tool CSP++. The translator is modified to support additional constructs from the language, and the object oriented application framework is enhanced with additional functionality that works alongside the new operations. An extended development paradigm is suggested, based on applying transformational methods from UML to C++. All techniques are demonstrated over three case-study examples.

Project Dissertation submitted to the University of Wales, Swansea
in Partial Fulfilment for the Degree of Master of Science
Declaration

This work has not previously been accepted in substance for any degree and is not being currently submitted for any degree.

September 12, 2012

Signed:

Statement 1

This dissertation is being submitted in partial fulfilment of the requirements for the degree of an MSc in Computer Science.

September 12, 2012

Signed:

Statement 2

This dissertation is the result of my own independent work/investigation, except where otherwise stated. Other sources are specifically acknowledged by clear cross referencing to author, work, and pages using the bibliography/references. I understand that failure to do this amounts to plagiarism and will be considered grounds for failure of this dissertation and the degree examination as a whole.

September 12, 2012

Signed:

Statement 3

I hereby give consent for my dissertation to be available for photocopying and for inter-library loan, and for the title and summary to be made available to outside organisations.

September 12, 2012

Signed:
Acknowledgements

We would like to thank the following for their support and direction, without whom this project would not have been successful:

- Dr Markus Roggenbach - Supervisor.
- Dr Anton Setzer - Second marker.
- Professor William Gardner - Originator of CSP++ at the University of Guelph, Canada.
- Professor Dr Holger Schlingloff of the Fraunhofer FIRST - Supervisor whilst in Germany.
- Professor Jon Holt - Representative of Atego, our industrial partner.

This opportunity cannot be missed to offer a special thanks to our family and friends, who made this project attainable and enjoyable, and for enduring our inevitable lamentation of the challenges we faced along the way.

This work is part-funded by the European Social Fund (ESF) through the European Union’s Convergence programme, administered by the Welsh Government.
## Contents

1 Introduction ................................................. 1
   1.1 Industrial partnership ................................ 1
   1.2 Document structure .................................... 2

I Background and existing technology ................... 3
   1.2 The selective formalism approach ...................... 4
   1.3 The Unified Modelling Language ...................... 4

2 The process algebra CSP .................................... 7

3 Motivation and related work ............................. 10

4 The existing CSP++ framework ......................... 12
   4.1 Analysis of CSP\_M data types ....................... 13
   4.2 Parameterised channels .............................. 17
   4.3 Support of CSP\_M data types in CSP++ ............... 17
   4.4 A CSP++ sample run ................................. 18
   4.5 Build and installation .............................. 20
   4.6 Architecture ........................................ 21
   4.7 Parsing system ...................................... 21
   4.8 The parse node hierarchy ......................... 24

II New ideas, more technology ........................... 27

1 Beginning modifications ................................. 28
   1.1 Identification in the source code ................... 30
   1.2 Sets .................................................. 31
   1.3 Sequences ............................................ 34
   1.4 CSP\_M functions .................................... 35
   1.5 User defined functions .............................. 36
   1.6 CSP++ preprocessor .................................. 37
   1.7 Handling the preprocessed file .................... 38

2 CPPUnit testing ............................................. 40

3 Demonstration .............................................. 42
   3.1 Children’s puzzle .................................... 43
   3.2 Linear sorting algorithm ............................ 46
   3.3 Escapology: The concrete coffin .................... 49

4 Evaluation .................................................. 53
   4.1 Scientific contribution .............................. 53
   4.2 Project management .................................. 54
   4.3 Future work ......................................... 54

5 Conclusion .................................................. 54
A Modified CSP\textsubscript{M} code for CSP++ input 57
B C++ code representing the translated ATM 58
C C++ output from CSP++ example translation 59
D CSP test scripts 63
E C++ code for the Children’s Puzzle 66
F C++ and UCF code for the linear sorting array 70
G C++ code for the concrete coffin example 76
H Review by industrial partner Atego 81
1 Introduction

The application of formal methods to software development is well established in academia, but not so well accepted in industry [Som96]. The reasons for this are often the esoteric mathematical notation used in formal specification algebras, the present lack of skilled individuals who can work with them, and the additional cost and time associated with producing a formal product.

Given the value of formal methods to the production of safe and secure systems, it makes sense to try and develop ways to make the techniques more widely available. This has been done for quite some time in incremental steps, with the introduction of interactive and even automated theorem proofs being one example of the advancing technology. In this document we look at another approach.

The result of research begun by Professor William Gardner in his PhD thesis [Gar00] is the CSP++ object oriented application framework. This work has since been continued by others, and is well on the way to becoming a fully fledged tool. Broadly, it allows the user to translate a formal CSP specification, about which one may have proved some properties, into ultimately executable C++ code. This C++ code is semantically equivalent to the CSP it was translated from, meaning that any properties proven there continue to hold in the C++ implementation.

Professor Gardner has kindly granted us the opportunity to begin work on the CSP++ tool, which will be the subject of this thesis. In its current state the translator works well, but has some areas that are still undeveloped. One of these areas are the data types which can be translated, as CSP\textsubscript{M} has a well defined set of types which are only partly represented in the translated C++ code. In this project, we will address this by satisfying the following aims:

- Continue support of primitive types.
- Add the following data types:
  - Sets.
  - Sequences.
  - Tuples.
- Support user defined functions.
- Demonstrate the use of these types in an example.
- Contribute regression tests for our new code.

As the types above are often used in all kinds of systems they are clearly worthwhile supporting, as it will increase the volume of products that can be developed this way. Furthermore, in a recent paper the original author of the tool notes that they would be a valuable improvement to the capability of the tool [Gar05]. The regression tests we provide will ensure that no existing functionality is lost, and help enhance the maintainability of the system for any future development. In addition, they will make sure that no new features we add conflict with anything that was previously added!

1.1 Industrial partnership

This project is being completed in partnership with the company Atego. They are “a world leading software tools and professional services company, focused on helping organizations engineer complex, mission- and safety-critical systems and software” [Ate12].

They are interested in this project because it aligns well with some of the product areas they are investigating. They wish to look particularly at the area of transformational development.
from requirements given in UML form, into some executable deliverable. With this aim in mind they have supplied us with one of the examples discussed in this thesis, where we investigate this development paradigm in more detail.

An appraisal of the cooperation from the point of view of Atego is included at Appendix H.

1.2 Document structure

As previously stated, CSP++ takes a CSP\(_M\) model as its input, and outputs a C++ program. The ideal case would be to support all data types in CSP\(_M\), and correctly map them to the equivalent in C++.

Part One of this document surveys the present state of transformational approaches to formal methods. We introduce some concepts and tools necessary for the rest of this document, and examine the CSP++ system in detail. In Part Two we begin modifications to the translator and its object oriented application framework, adding functionality in accordance with the aims stated above. Finally we offer some closing remarks, reviewing what was achieved and what remains to be done to continue development in this area. In general, everything we present in Part One is an examination of work done by others. All of the work presented in Part Two can be regarded as our contribution to the project.
Part I

Background and existing technology
1.2 The selective formalism approach

When one thinks of a formal system, one typically thinks of an entire system being rigorously specified using some mathematical model. This results in a very precise and useful artefact, but creating it is often a long, costly and laborious task. The primary focus of CSP++ is the development of concurrent systems. This is because a great deal of complexity arises from the communication between two or more processes [MK06]. A secondary focus was to make the process easier, “an unabashed attempt to breech the resistance of software developers to adopting pure formal methods” [Gar05]. This has been achieved by allowing user coded functions to be added to the translated software. The framework allowing this introduced in [CXG05] involves formally specifying and verifying a so called ‘backbone of control’ in a concurrent system. This backbone must be in sole control of all interprocess communication in the entire system. Because the user coded functions have not been formally verified, their functionality is subject to some restrictions [Gar00]:

- Events on which processes synchronise cannot have user code attached.

- User code must not engage in any interprocess communication, as this would ‘go behind the back’ of the verified CSP backbone.

There are some very good reasons for adopting this design methodology. As previously mentioned, expertise in formal systems is not always widespread, or is its employment necessarily cost effective in a business situation. By reducing the amount of functionality that has to be expressed formally, the amount of man hours required is reduced in turn. Another good reason to adopt selective formalism in industry is the naturally changing levels of integrity required. Consider a common application of formal methods: the aerospace industry. The parts of software that are flying the aircraft must be verified, in fact it is mandatory to do so, because they are truly safety critical - their failure could introduce a risk to life. However, failure of the software being used to show the in flight movie will at worst introduce boredom amongst the passengers, so need not be subject to the lengthy and costly verification process. Some aspects of software are very difficult to express in a formal algebra too. Processes such as interaction with database management systems and printers are often complex, but relatively unimportant. These can be best left to traditionally tested functions.

Selective formalism is not proposed as a catch all solution though. As already stated, the majority of complexity arises from inter-process interaction, and this remains verified. In fact, for some practitioners who already produce software verified in its entirety, selective formalism may represent a retrograde step in the quality of their products.

1.3 The Unified Modelling Language

The Unified Modelling Language, or UML, is a standard introduced in 1997 to represent the architecture of a software based system. It is now coordinated by the Object Management Group, going through several revisions to become what is now the industry standard in software modelling [UML12].

Before continuing with a survey of the UML, we will first place it within the context of this project. We are working on extending the tool CSP++, which allows us to carry out automatic synthesis of a model in the language CSP, to an executable program in the language C++. This is a very useful step in itself, but is necessarily constrained by the level of granularity imposed in a CSP model. These models are typically relatively low level, expressing the processes in the detail that is required for verification techniques to work. The UML allows a higher level model to be described, in which systems engineers can capture real world concepts in a way that can
be manipulated in the computer. It is too high a level to be used as a robust formal methods tool, but does have the capability to itself be translated, with CSP being amongst the target languages. This opens up an extended line of development, in which a developer can model a system in the UML, carry out verification on it in CSP, translate it to C++, and finally link it with user coded functions.

Figure 1: The proposed development paradigm using CSP++ with other tools.

The best way to understand a graphical language like the UML is in a graphical form, so a hierarchy of UML objects is shown in Figure 2. In this standard, objects are grouped in two, structural and behavioural diagrams. The structural diagrams present a static view of the objects, such as what classes are present, and what attributes they have. The behavioural diagrams represent the dynamic aspects of the system, such as communication or relationships between objects.

Figure 2: Objects in UML 2.2. [Gro09].

Figure 2 makes clear the split between structure and behaviour diagrams, as well as showing the different cases of each sub-diagram.

A team from the University of Leicester carried out a case study, outlining a theoretical method to formally transform a UML diagram to a CSP script [BEH]. They began by constructing a pair of meta-models, that is a model representing the UML and the CSP script, to simplify each enough to allow the transformation to happen, whilst preserving the generality. Using these meta-models, they give a transformation method by defining some rules which identify seven equivalences between fragments of UML diagrams and CSP statements. An example of this is a decision node in the UML, as shown in Figure 3. The UML diagram fragment is
shown on the left, and the equivalent CSP on the right. Note that in the meta-model of CSP code, the $\not\triangleright$ symbols are a representation of an if then statement.

![Diagram](image1)

Figure 3: Transformation rule for a UML decision node [BEH].

They did not create an implementation of the process, but they did produce an example in the case study. We will restate the example here, and then illustrate our proposed design flow by taking it a step further, translating it using CSP++ to executable C++ code. The UML activity diagram for the example is shown in Figure 4, representing the process a controller might carry out to organise a service with the driver of a remote vehicle. The equivalent CSP code is shown in Listing 1, where we have manually converted it from the meta-model syntax to CSP$_M$ to allow it to be processed further.

![Diagram](image2)

Figure 4: Activity diagram used in the example [BEH].

The diagram begins with a server receiving an alert, so a process is defined that carries out
Listing 1: CSP code resulting from the transformation of Figure 4.

\[
\begin{align*}
S1 &= \text{serverReceiveAlert} \rightarrow S2 \\
S2 &= \text{getDriverPhoneData} \rightarrow S3 \\
S3 &= \text{callDriver} \rightarrow S4 \\
S4 &= \text{if} (\text{nohelp}) \text{ then } M1 \text{ else } (\text{if} (\text{askhelp}) \text{ then } D2 \text{ else } D1a ) \\
M1 &= C1 \\
D2 &= DM \\
D1a &= \text{assessDescription} \rightarrow D1b \\
D1b &= DM \\
DM &= \text{if} (\text{real}) \text{ then } D3 \text{ else } M2 \\
M2 &= C1 \\
C1 &= \text{cancelAlert} \rightarrow C2 \\
C2 &= \text{SKIP} \\
D3 &= F1 \parallel F2 \parallel F3 \\
F1 &= \text{getMapLocation} \rightarrow J1 \\
F2 &= \text{processAlert} \rightarrow J2 \\
F3 &= \text{getServiceFormat} \rightarrow J3 \\
J1 &= \text{processJoin} \rightarrow E1 \\
J2 &= \text{processJoin} \rightarrow \text{SKIP} \\
J3 &= \text{processJoin} \rightarrow \text{SKIP} \\
E1 &= \text{createServiceDescription} \rightarrow E2 \\
E2 &= \text{SKIP}
\end{align*}
\]

this action. The subsequent actions of getting the drivers phone data and beginning a call are chained sequentially from the first process. It would be trivial to condense these actions, but at this stage the authors have not put any work into optimising the process, so we leave it at this representation. At the stage S4, the flow of control branches, so a process is defined using an if then construct. If no help is available the alert is cancelled, otherwise the description is assessed or help is sought. After this stage some concurrency is introduced, represented by the thick horizontal line in the diagram, allowing the location to be found, the alert processed and the service format be checked all at the same time. Once all three have terminated the control flows to stage E1, where the create service description action is carried out, and the system successfully terminates, a CSP \text{SKIP} action.

The CSP\text{M} code can then be used as input to the CSP++ translator - though not without some small changes made by hand, such as adding channel declarations and instantiating variables with values in the conditional statements. The code following changes is shown in Appendix A for the interested reader.

Before it is translated, we can simulate the model using the tool ProBe, ensuring that it is semantically correct. Any verification could be done at this stage too, for instance checking for determinism, and the absence of live or dead locks. The model in ProBe is shown in Figure 5.

Once all of the formal simulation and verification is complete, the code can be run through the CSP++ translator, and compiled alongside the object oriented application framework. The code output gives a strong example of the potential of the selective formalism paradigm: the parallelism has been verified so there can be no hidden problems there, and the events allow additional user coded functions to add functionality to the system. The full translated code can be found in Appendix C, the user session in the terminal is shown in Figure 6.

2 The process algebra CSP

CSP stands for Communicating Sequential Processes, it is an algebra designed for use in modelling systems involving concurrency. It was developed by Tony Hoare, who wrote the classic
text describing it in 1985. This has been most recently revised in 2004 [Hoa04]. At its inception CSP was only a ‘blackboard language’, that is it was only used to model and prove properties manually. As the field evolved it became advantageous to have CSP running on a computer, so that the work done could be automated. This gave rise to some differing syntax, as the mathematical notation previously used cannot be directly represented in an ASCII text file that a compiler can read.

In CSP, we reason about processes and events. A process is some sequence of events; an event is something that one might observe in real life. Events can involve input or output. Here we make a small example to demonstrate the basics of CSP. The alphabet $A$ of a process $P$, $\alpha(P)$, is the set of all of the events that $P$ can carry out. Knowing what an alphabet is, we can consider the simplest definition of a process. Let $a \in A$ be an event, and $P, P'$ be processes. Where $P \xrightarrow{a} P'$, the process $P$ carries out an event $a$, and becomes the process $P'$. In CSP, the machine readable version of CSP, this process would be represented as $P = a \rightarrow P'$. In CSP events are considered to be atomic and instantaneous.

Processes can be defined in terms of themselves by using recursion. The standard example of a recursive process is of a clock that can tick infinitely. To model this, we write a process $CLOCK$, with $\alpha(CLOCK) = \{\text{tick}\}$. The process definition is then $CLOCK = (\text{tick} \rightarrow CLOCK)$. This states that the process can carry out a tick action, then continue to act as the process $CLOCK$. Note that all processes can be expressed as an equivalent labelled transition system. The labelled transition system for $CLOCK$ is shown in Figure 7.

The key strength of CSP for modelling concurrent systems comes from its ability to easily
synchronise two or more processes on some event or events. Let’s take the example of two processes, one which sends some item of data, and one which receives it. Both processes can execute their actions independently until the stage at which they need to communicate, when they must do so synchronously. The two processes are given below:

\[
\begin{align*}
\text{SEND} & = \text{getData} \rightarrow \text{prepareToSend} \rightarrow \text{transfer} \rightarrow \text{done} \rightarrow \text{SKIP} \\
\text{RECEIVE} & = \text{prepareToReceive} \rightarrow \text{transfer} \rightarrow \text{save} \rightarrow \text{done} \rightarrow \text{SKIP}
\end{align*}
\]

Note some of the conventions in use here. Processes are given upper case names, events begin lower case, and \text{SKIP} is the action carried out when a process terminates successfully. To execute these two processes, we put them together in parallel composition. In CSP\textsubscript{M} it is expressed thus:

\[
\text{SEND} \{ | \text{transfer} | \} \text{RECEIVE}
\]

This states that \text{SEND} and \text{RECEIVE} run in parallel but independently, until they must synchronise on their transfer actions.

Unlike other popular languages such as C, no standard has ever been published for CSP, so if one wishes to, new functionality can be added at will. As a result of this, other dialects of CSP have been developed, often aimed at solving the authors own specific problems. Early editions of CSP++ used the csp12 dialect as their input; one reason for this was that csp12 was developed in the same institute. This soon became a limitation to the development of CSP++ as there was insufficient tool support for the dialect to make it useful. In version 4.0 the designers chose to switch to use CSP\textsubscript{M} as the input language [Gar05]. CSP\textsubscript{M} has been much more widely used, and good tool support exists, provided by a company called Formal
systems (Europe) [For12]. They market some tools for verification, exploring the structure of processes and checking syntactic correctness.

It must be acknowledged that there are other algebras similarly suited to verifying concurrent systems. In principle any language that allows textual representation and verification of properties will satisfy the requirements of an input language. An example of an alternative is CCS [Mil95].

The principle reasons stated by Professor Gardner for adoption of CSP were the availability of in house tools and expertise, as well as his own familiarity with the language [Gar00]. We argue that since only the dialect was later changed to acquire better tool support, the choice of language was a good one since such tools are now well established for CSP.

3 Motivation and related work

This section is dedicated to providing some motivation for formal methods in general, and for the use of automatic instead of manual translation from model to executable implementation.

To provide this motivation, we move the reader’s perspective out of the research environment into the real world. The International Space Station was first launched on the 31st October 2000, and took with it a fault tolerant computer. This machine was made up of several redundant sets of hardware each carrying out the same computations, so that accurate results could be depended on despite the hardware problems caused by the presence of increased radiation in space.

In order to detect which parts of the redundant system were returning incorrect answers, the designers implemented the Byzantine agreement protocol. This protocol was first identified in 1982 as a way to allow correct processors to reach a common decision without being effected by erroneous or malicious input from rogue processors [LSP82].

The analogy given in the original paper is of a Byzantine army approaching the enemy. The army is made up of several brigades each led by a general. Some of the general’s remain loyal, but there are a minority of traitors. All of the generals must agree on a plan to attack the enemy, but the traitors will try and cause the loyal generals to reach an incorrect decision. If one implements the Byzantine agreement protocol we assume that the loyal generals act as the algorithm instructs them to, while the traitors may act arbitrarily. Carried out correctly, this will allow all loyal parties to come to the correct agreement.

It can clearly be seen how this applies to detecting a faulty processor in the space stations fault tolerant computer. The engineers working on the project manually encoded the protocol in the language Occam, and successfully tested it. The work done in [BKPS97] took the Occam implementation, and manually constructed a CSP model of it. They then verified the system, seeking any live lock or dead lock situations. Where problems were detected, they were traced back to the Occam code, and both that and the model updated.

By the end of their work, the team reported finding “seven deadlock situations” and “about five live locks” in the original Occam software [PB99].

Consider now Figure 8. This diagram is effectively made up of two parts, both of which start in the middle with the problem statement and production of pseudo-code. The original team working on the space station moved to the left, implementing the pseudo-code in Occam. From that implementation, the team in [PB99] manually abstracted the CSP code, carried out the model checking, and refined the code as they found problems.

The method proposed by CSP++ is represented on the right hand side of the diagram. Here, instead of manually implementing and then abstracting, the pseudo-code is manually encoded straight to CSP, and model checked. Once all required properties are satisfied, the CSP can be automatically translated into, in the case of CSP++, an executable C++ implementation.
It can be seen from the foregoing that robust tool support for this new approach would have cut down the development and refinement time that the Occam system underwent. The safety critical nature of the application is clear too, and the deadlocks and live locks found in themselves justify the use of formal methods.

Another key question for our method is how far one can push the concurrent aspect of our programming. This is especially relevant while CSP is used as an input language, because of its inherent parallelism. Presently, the resulting applications are compiled using the POSIX ‘pthreads’ library, which allows interprocess communication and sharing of local memory resources. To really be able to expand, one would need to consider running applications in a distributed way, across several machines which do not share memory in the same physical address space. Some work has been done to make the use of POSIX threads a little easier by implementing a thread pool [Kri04], some other research that was not carried out so recently but shows more direct promise for distributed computing was in [Mue97]. Here the author proposed and showed a new runtime which was implemented on top of POSIX pthreads, which were dubbed distributed shared memory (DSM) threads. Systems built using POSIX threads could be ported to a DSM model with minimal changes to their original implementation, allowing them to take advantage of the distributed environment.

CSP++ is not the only potential solution to the problem of program extraction from specifications. Another very similar approach comes from the University of Kent, called JCSP [WBM+07]. This is a framework based upon Java, in which the developer writes statements of Java code that represent the CSP specification. This has several advantages and disadvantages. The first can be considered to be positive or negative: one writes in Java like statements, rather than in pure CSP. From a positive point of view this reduces the learning curve for a new developer, making it easier to acquire a skilled team to work with the methodology. On the negative side, this pseudo-CSP cannot be directly model checked as it is not written in a syntax that tools such as FDR2 can interpret. Model checking can still be carried out, but would need manual abstraction to pure CSP before formal techniques could be applied. Note that this is very similar to the work done for the space station, illustrated in Figure 8.

A clear advantage of JCSP is that because it is built on top of, and by using existing Java thread models, it can run in a distributed environment without any changes at all. However, it must always run on top of a Java virtual machine, which means that when distributed every instance of it must have a virtual machine too. As the size of the network grows, so would the
resource footprint associated with it.

A second related system revolves around model based design of concurrent programs [MK05]. In this approach they model systems as finite state machines, often describing them with a formal CSP like notation. We noted in Section 2 that CSP process can be equivalently represented as a finite state machine, which means in principal they too can undergo model checking - although tool support would be needed for whatever representation is in use. From the model though, their approach differs to the CSP++ one. They point out that because models “capture only some aspects of a program, we do not attempt to automatically generate entire concurrent programs” [MK05]. Instead, they offer a set of rules that can be applied manually to create a Java implementation equivalent to the specification. It is interesting to note that they report the results of implementing some examples published in the paper, in which four bugs were found, two of which were as a result of incorrectly applying their manual rules.

CSP has two clear advantages over these approaches, first that it is automatic, and second that it uses CSP_M as its input language. This removes the possibility of manual errors, and gives the user the opportunity to exploit good tool support. Although some aspects that the alternatives already carry out well need some work to apply in CSP++, they are not beyond the scope of future work.

4 The existing CSP++ framework

This section will be concerned with the structure and behaviour of the existing CSP++ system. We cannot expect to extend it without having a thorough understanding of how it has been designed and how it works, so here we address that need. We begin by showing an example of the system in action. We then look into the details of its implementation, both the theory and practice by which it works.

The CSP++ system is an object oriented application framework (OOAF) that translates specifications written in CSP_M to C++ source code, which can be compiled to an executable with any standard C++ compiler. The concept of OOAFs is not new, the first serious review of them was carried out in 1997 by the ACM [FS97]. It defined an OOAF as a “reusable, ‘semi-complete’ application that can be specialized to produce custom applications”. They listed benefits of code re-usability, modularity, and especially extensibility. As each implementation produced by the CSP++ framework will effectively extend the runtime environment, creating it as an OOAF makes sense.

An important pair of decisions to explore is why an OOAF was chosen, and why it was implemented in C++. The key reason for using an OOAF is that it will become the code generation target. Figure 10 shows some possible targets, including an OOAF.

Going from the bottom of the image, we consider assembly language as the first code generation target candidate. Here, the complexity of the code to be generated is very large, because there are a lot of memory and register aspects to be taken care of. Moving up another level to high level language (HLL) source code, the task becomes a little easier. This time, the register allocation and some of the memory concerns are taken over by the runtime (RT) facilities of the operating system (OS). The next level up is the Java source code; once again this represents a level of difficulty lower. The top level, where the OOAF is the target is the easiest target to reach. Note too that it is shown as a smaller block than the Java Virtual Machine (JVM). This is because the OOAF is a system dedicated to one purpose, meaning it has a far smaller resource footprint than the general purpose JVM.

C++ was chosen as the implementation language for several reasons [Gar00]:

- Object orientation. Clearly, an OOAF should be implemented in a language which does
support object orientation. C++ provides us with the facilities we require.

- Compilation to assembly language. The standard C++ compilers produce assembly language code as their output. Not only does this take care of the complexity involved, it removes the requirement for a virtual machine, and the overheads associated with one.

- Popularity. Recall that user coded functions are to be added on to the translated code. C++ is a good choice here, as already many developers are qualified in its use.

Professor Gardner did identify some key disadvantages of using C++. It has a relatively large resource requirement, meaning code generated may not be suitable for some resource critical applications such as embedded systems. By default, it does not support a good model of multithreading. This is easily overcome by using additional libraries, such as POSIX ‘Pthreads’. The name ‘POSIX’ is the result of a call for a memorable name, and stands for “Portable Operating System Interface”. It is a standard provided by the IEEE since 1988, that is actually much broader than just Pthreads. The standard describes itself nicely, by stating that it “defines a standard operating system interface and environment to support application portability at the source-code level” [IEE90]. In practice the Pthreads library provides a nice way to implement threading models within source code, that is portable within UNIX like execution environments. Recall that a thread is a lightweight path of execution within a process. There are many advantages to the use of threads, one being enhanced speed of execution. This is because it does not take long to switch between threads compared to processes, because all the threads within a process share the same address space, stack and memory. The performance increase is even better when multiple processors are available. There is cost associated with multi threading of course, such as the resource overhead related with synchronisation. Debugging can be more difficult too as a result of the complexity of a concurrent implementation - but this is one of the aspects selective formalism and code generation sets out to overcome [But97].

We feel that Professor Gardner was justified in reporting that these drawbacks did not outweigh the advantages of it being selected [Gar00].

4.1 Analysis of CSP\textsubscript{M} data types

This part of the document is dedicated to analysing what is defined in CSP\textsubscript{M}. The information for this is drawn from the work of Scattergood [Sca98], where he presents the denotational
semantics of CSP\(_M\). Note that comparison operators and complex types have what is known as a base type. This base type is inherited from the types that it is instantiated with, that is the arguments of a comparison operator, or the type of objects stored in a complex type. Where the type is inherited from a base type, we note the type of that object as ‘\textit{type}, a predefined type’. Where we feel that an operation needs some explanation it is either noted directly after its profile, or in the following paragraph.

4.1.1 Comparison operators

In addition to the operators defined over each type given below, the following operators are defined across all types. Because of the diversity, for instance between a pair of boolean values and a pair of sets, they are evaluated differently depending on the types they are instantiated with.

**Alphabet:** \textit{type}, a predefined type.

<table>
<thead>
<tr>
<th>Operations</th>
<th>Signature</th>
</tr>
</thead>
<tbody>
<tr>
<td>==</td>
<td>\textit{type} \times \textit{type} \rightarrow boolean</td>
</tr>
<tr>
<td>!=</td>
<td>\textit{type} \times \textit{type} \rightarrow boolean</td>
</tr>
<tr>
<td>&lt;</td>
<td>\textit{type} \times \textit{type} \rightarrow boolean</td>
</tr>
<tr>
<td>&gt;</td>
<td>\textit{type} \times \textit{type} \rightarrow boolean</td>
</tr>
<tr>
<td>≤</td>
<td>\textit{type} \times \textit{type} \rightarrow boolean</td>
</tr>
<tr>
<td>≥</td>
<td>\textit{type} \times \textit{type} \rightarrow boolean</td>
</tr>
</tbody>
</table>

Equality is defined over all types except those containing processes or lambda terms. Ordering operations are defined on elements of sets, sequences and tuples, but not on booleans or
user defined types.

4.1.2 Numbers

Although this section is referred to as numbers, note that CSP\textsubscript{M} only officially supports integers. Floats are present in the language, but are not supported as they have only been introduced experimentally. For this reason we do not include them here.

**Alphabet:** $\mathbb{N}$, the space of integer literals.

<table>
<thead>
<tr>
<th>Operations</th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>+</td>
<td>number $\times$ number $\rightarrow$ number</td>
</tr>
<tr>
<td>$-$</td>
<td>number $\times$ number $\rightarrow$ number</td>
</tr>
<tr>
<td>$-$</td>
<td>number $\rightarrow$ number</td>
</tr>
<tr>
<td>$\times$</td>
<td>number $\times$ number $\rightarrow$ number</td>
</tr>
<tr>
<td>$\div$</td>
<td>number $\times$ number $\rightarrow$ number</td>
</tr>
<tr>
<td>$%$</td>
<td>number $\times$ number $\rightarrow$ number</td>
</tr>
</tbody>
</table>

Note here the overloading applied to the ‘$-$’ operator, representing subtraction and unary minus.

4.1.3 Booleans

**Alphabet:** $\{true, false\}$

<table>
<thead>
<tr>
<th>Operations</th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>$\land$</td>
<td>boolean $\times$ boolean $\rightarrow$ boolean</td>
</tr>
<tr>
<td>$\lor$</td>
<td>boolean $\times$ boolean $\rightarrow$ boolean</td>
</tr>
<tr>
<td>$\neg$</td>
<td>boolean $\rightarrow$ boolean</td>
</tr>
</tbody>
</table>

Exclusive or is not defined as a language operation, but it could be used in the language by constructing it from the above operators. Implication is not included here, because it is a language construct that merely involves the evaluation of boolean expressions: \texttt{if <expression> then <expression> else <expression>}, where each expression contains only the operators given above.

4.1.4 Sets

A set is an unordered collection, with elements all of the same type. Repeat elements are not allowed.

**Alphabet:** \textit{type}, a predefined type.
### Operations:

<table>
<thead>
<tr>
<th>Operation</th>
<th>Signature</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>union(a, b)</td>
<td>set × set → set</td>
<td>union of two sets.</td>
</tr>
<tr>
<td>inter(a, b)</td>
<td>set × set → set</td>
<td>intersection of two sets.</td>
</tr>
<tr>
<td>diff(a, b)</td>
<td>set × set → set</td>
<td>performs set minus, evaluating a \ b.</td>
</tr>
<tr>
<td>Union(a, b)</td>
<td>set × set → set</td>
<td>distributed union of two sets.</td>
</tr>
<tr>
<td>Inter(a, b)</td>
<td>set × set → set</td>
<td>distributed intersection of two sets.</td>
</tr>
<tr>
<td>concat(a, b)</td>
<td>set × set → set</td>
<td>concatenates two sets.</td>
</tr>
<tr>
<td>member(x, b)</td>
<td>type × set → boolean</td>
<td>tests for membership of an element in a set.</td>
</tr>
<tr>
<td>card(a)</td>
<td>set → number</td>
<td>calculates cardinality (number of elements) in the set.</td>
</tr>
<tr>
<td>empty(a)</td>
<td>set → boolean</td>
<td>tests if the set is empty.</td>
</tr>
</tbody>
</table>

Note the two versions of union and intersection. The sentence case version represents distributed union and intersection, which is applied where a set is comprised of other sets. For example, \(\text{union}\{\{a, b, c\}, \{c, d, e\}\} = \{a, b, c, d, e\}\), whereas \(\text{Union}\{\{a, b\}, \{b, c\}, \{c\}\}\) = \(\{a, b, c\}\).

Because sets do not allow the repetition of elements, only types where equality is defined can be placed in a set. This disallows the instantiation of sets containing processes or lambda terms.

#### 4.1.5 Sequences

Mathematically, a sequence is defined very similarly to an array, but has some fundamental differences in the way it can be used. A sequence is a contiguous, homogeneous structure, meaning all of its elements are of the same type. This is known as the base type of the sequence. Although every sequence has a finite length, unlike an array it is flexible. In this respect it is closely analogous to a dynamic array such as the `ArrayList` in Java. Unlike any array though, a sequence is not random access, that is one cannot directly access any element in the sequence at any arbitrary time [Wir85]. Instead we rely on sequential operators such as head and tail to process the list and get to the element we require.

**Alphabet:** `type`, a predefined type.

**Operations:**

<table>
<thead>
<tr>
<th>Operation</th>
<th>Signature</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>length(s)</td>
<td>sequence → number</td>
<td>gets the length of sequence s</td>
</tr>
<tr>
<td>null(s)</td>
<td>sequence → boolean</td>
<td>tests if a sequence is empty, that is null(s) ≡ s == &lt;&gt; .</td>
</tr>
<tr>
<td>head(s)</td>
<td>sequence → type</td>
<td>returns the first element of s, providing s is non-empty.</td>
</tr>
<tr>
<td>tail(s)</td>
<td>sequence → type</td>
<td>returns all but the first element of a non-empty sequence s.</td>
</tr>
<tr>
<td>concat(s)</td>
<td>sequence → sequence</td>
<td>joins a sequence of sequences.</td>
</tr>
<tr>
<td>elem(x, s)</td>
<td>sequence × type → boolean</td>
<td>tests if element x occurs in sequence s.</td>
</tr>
</tbody>
</table>

#### 4.1.6 Tuples

A tuple is an ordered list of heterogeneous elements, with some cardinality \(n\). This is the reason they are often referred to as ‘\(n\)-tuples’, where the \(n\) denotes the number of elements in the tuple. In \(\text{CSP}_M\), they are delimited with parentheses and commas, for instance \((2, 3, 4, 1)\) is a 4-tuple.

**Alphabet:** `type`, a predefined type.
Operations:

CSP\(_M\) does not define any operations over the tuple data type, though they will inherit the operations of its base type. Where a tuple is used in a function call, for example \texttt{multiply((x, y))}, note the extra set of brackets. These are required to ensure that the function multiply treats the tuple as a single argument.

4.1.7 User defined functions

In CSP\(_M\), a user can define their own functions by using operations and expressions already available. An example is the user defining a new function to increment an integer, shown in Listing 2.

Listing 2: A user defined type in CSP\(_M\).

\begin{verbatim}
channel begin, end
increment (n) = n + 1
Proc (i) = if (increment (i) == 2) then begin else end
\end{verbatim}

These user defined types are analogous to a method in a traditional programming language, because they allow an easy way to define a module that carries out some processing. The constructs used in the definition can be nested, such as would occur if more than one branching statement was used.

4.1.8 Lambda abstractions

Lambda calculus is a mathematical notation developed by Alonzo Church. It is based on functions, and generalising expressions by giving them names, variables and evaluating them [Mic89]. It enforces a strict prefix notation, so for example the expression \(\sin(x) + 4\) would be expressed as \(+ (\sin x) 4\). Along with some small additions, lambda calculus goes together to make up the functional programming languages we are familiar with today [Jun04].

CSP\(_M\) does support the expression of such lambda abstractions. We do not expect offering support for such expressions to be within the scope of our thesis, so for now refrain from giving them any further treatment.

4.2 Parameterised channels

Though not a data type, we include parameterised channels here because they are a useful feature of CSP\(_M\). This is a mechanism by which one can attach some integer value to a channel, and perform arithmetic on that value.

CSP++ supports the use of such channels, but bugs have been reported of it not successfully translating them when they are used as a synchronising action in parallel composition. We have not yet been able to reproduce this bug, but will continue to test for it and if the problem is found, produce a bug fix as stated in our aims.

4.3 Support of CSP\(_M\) data types in CSP++

To investigate what types and operations can be translated by CSP++ in its current version, we have made a series of experimental tests. Each one of these tests is a process that exercises a data type and an operation over it. Formal Systems (Europe) provide a tool \texttt{Checker} [For12], which we use to ensure each test is syntactically correct with respect to CSP\(_M\), and if it is, we translate it with \texttt{cspt}. The source code output is compiled, and run. If the program behaves
Listing 4: ATM CSP specification.

channel enterPin : \{1000..9999\}
channel setPin : \{1000..9999\}
channel dispenseCash, rejectPin, pinOK

\[\text{ATM} = \text{setPin}\?y \rightarrow \text{enterPin}\?x \rightarrow \text{if} \ (x == y) \]
\[\text{then} \]
\[\quad \text{pinOK} \rightarrow \text{dispenseCash} \rightarrow \text{ATM} \]
\[\text{else} \]
\[\quad \text{rejectPin} \rightarrow \text{ATM} \]

\[\text{SYS} = \text{ATM} \]

Listing 5: Commands and output of the translated ATM.

daniel@EeePC:~/Desktop$ cspt CSPTranslate.csp
Reading from file: CSPTranslate.csp
Translating to file: CSPTranslate.cc
daniel@EeePC:~/Desktop$ g++ -o ATM CSPTranslate.cc -lcspxx -lpth -w
daniel@EeePC:~/Desktop$ ./ATM

as expected and correctly evaluates the operation under test, it is considered to have passed, otherwise it has failed.

To illustrate this, we show an example test in Listing 3, and the output associated with it. This test is for integer multiplication. This is an important table as it sets out the work to be done in the project. It can be seen that the integer, boolean and user defined types are fully supported, but sets, sequences, tuples and lambda abstractions all fail.

Listing 3: Example test run

channel test : Int
channel echoBack : Int
SYS = test\?x \rightarrow \text{echoBack}!x \ast x \rightarrow \text{SKIP}

daniel@EeePC:~/Desktop ./test
Channel: test: Enter integer: 5
Channel: echoBack!25
Pass

To make sure our tests were fair, each one was executed in ProBe manually, in such a way as we achieved both a pass and a fail result. This leaves us satisfied that the behaviour we are testing for is present and correct in CSP\(_M\), so it only has to be seen if it is in CSP++. Table 1 shows a summary of each test carried out, and the result we obtained.

4.4 A CSP++ sample run

Here we will make a small example of the system translating a CSP specification into an executable program. The CSP source code is given in Listing 4, and it represents a very simple automated teller machine (ATM). The machine accepts a PIN, checks if it is correct against one set by the user, and if it is, dispenses some cash.

To make the translation and run the program, we issue the series of commands shown in Listing 5. The cspt command runs the CSP++ translator, changing the CSP into C++. The g++ line simply invokes the GNU C++ compiler, producing the executable ATM.
Table 1: Results of tests on CSP++

<table>
<thead>
<tr>
<th>Data type</th>
<th>Operation</th>
<th>Result</th>
</tr>
</thead>
<tbody>
<tr>
<td>Integer</td>
<td>Multiplication</td>
<td>Pass</td>
</tr>
<tr>
<td>Integer</td>
<td>Unary minus</td>
<td>Pass</td>
</tr>
<tr>
<td>Integer</td>
<td>Addition</td>
<td>Pass</td>
</tr>
<tr>
<td>Integer</td>
<td>Subtraction</td>
<td>Pass</td>
</tr>
<tr>
<td>Integer</td>
<td>Division</td>
<td>Pass</td>
</tr>
<tr>
<td>Integer</td>
<td>Equality</td>
<td>Pass</td>
</tr>
<tr>
<td>Integer</td>
<td>$&lt;$, $&gt;$</td>
<td>Pass</td>
</tr>
<tr>
<td>Integer</td>
<td>$\leq$, $\geq$</td>
<td>Pass</td>
</tr>
<tr>
<td>Integer</td>
<td>Modulo</td>
<td>Pass</td>
</tr>
<tr>
<td>Boolean</td>
<td>Equality</td>
<td>Pass</td>
</tr>
<tr>
<td>Boolean</td>
<td>And, Or</td>
<td>Pass</td>
</tr>
<tr>
<td>Boolean</td>
<td>Not</td>
<td>Pass</td>
</tr>
<tr>
<td>Set</td>
<td>Union</td>
<td>Fail</td>
</tr>
<tr>
<td>Set</td>
<td>Intersection</td>
<td>Fail</td>
</tr>
<tr>
<td>Set</td>
<td>Difference</td>
<td>Fail</td>
</tr>
<tr>
<td>Set</td>
<td>Distributed union</td>
<td>Fail</td>
</tr>
<tr>
<td>Set</td>
<td>Distributed intersection</td>
<td>Fail</td>
</tr>
<tr>
<td>Set</td>
<td>Member</td>
<td>Fail</td>
</tr>
<tr>
<td>Set</td>
<td>Cardinality</td>
<td>Fail</td>
</tr>
<tr>
<td>Set</td>
<td>Empty</td>
<td>Fail</td>
</tr>
<tr>
<td>Sequence</td>
<td>Length</td>
<td>Fail</td>
</tr>
<tr>
<td>Sequence</td>
<td>Null</td>
<td>Fail</td>
</tr>
<tr>
<td>Sequence</td>
<td>Head</td>
<td>Fail</td>
</tr>
<tr>
<td>Sequence</td>
<td>Tail</td>
<td>Fail</td>
</tr>
<tr>
<td>Sequence</td>
<td>Concatenation</td>
<td>Fail</td>
</tr>
<tr>
<td>Sequence</td>
<td>Element membership</td>
<td>Fail</td>
</tr>
<tr>
<td>Tuple</td>
<td>Declaration</td>
<td>Fail</td>
</tr>
<tr>
<td>User defined</td>
<td>Use of user defined function</td>
<td>Fail</td>
</tr>
<tr>
<td>Lambda abstraction</td>
<td>Use of lambda function</td>
<td>Fail</td>
</tr>
<tr>
<td>Parameterised channels</td>
<td>Use of parameterised channel</td>
<td>Pass</td>
</tr>
</tbody>
</table>
Listing B in Appendix B shows the C++ code that was produced from the CSP specification. It can be seen how functions are generated that represent the processes, the choice is represented by an if ... then structure, and parameters can be read in. Finally, the running program can be seen in the terminal screen shot in Figure 11.

![Figure 11: The running ATM program.](image)

### 4.5 Build and installation

The original research group has kindly given us access to a fork of the source code on their SVN repository. The tool was developed using Linux, and for reasons of compatibility we will continue to use the same system - Ubuntu 12.04, in this case. Checking the code out from the repository gives a directory with configuration and make files, as well as a Doxyfile used to set up documentation software. The source code itself is organised in sub-folders, and consists of 129 items.

The process to build and install CSP++ is all done using the command line interface. We began by running a tool called *automake*. This is a Linux tool that generates a make file from a *Makefile.am* file. Having resolved any dependencies, one can run a second tool called *autoconf*, which creates a configuration script from the provided *configuration.in* file.

Having prepared the files, one can now run the generated configuration script, then the make script. It is worth noting that when one runs the configuration script, supplying a `--prefix=<path>` option will allow you to install the system to a directory of your choice. On running make we encountered an error, where the file `iostream.h` is included in a sub-make file. This is a standard library, so we are unsure why it is explicitly included here, and will follow it up with the research group. The problem was fixed by removing the include statement from the file, and the make run completes successfully. Running ‘sudo make install’ completes the installation and copies it to the specified directory.

Translating `.CSP` files and compiling them is now trivial. At the command line one invokes `cspt file.csp` and the C++ source is produced. This can be compiled using any standard C++ compiler, providing one uses the `-lcspxx -lpthread` flags when it is run.
4.6 Architecture

This section is dedicated to an investigation of how the CSP++ translator actually works. It should be made clear that the software is made up of two separate parts, the translator and the OOAF. The translator is the executable part of the system, and is solely responsible for transforming the CSP input file into a C++ source code output file. All of the source code for this is in a separate xlator folder. The OOAF is not executable, rather it is linked at compile time with the translated C++ code. Without this linking the definitions that have been generated in the translated source are not defined, so it cannot compile. The OOAF can almost be thought of as a very small virtual machine, as it adds ease of use functionality to the code, but is itself compiled by the C++ compiler to form the final executable.

Because we do not make modifications, only additions, to the OOAF, we do not present a comprehensive study of it here. We begin by examining a parsing system, as this is what is at the heart of any language processor. From there we will look into the architecture of the software itself, and how the classes are laid out. This should establish a picture of what areas are complete, and which are the ones we will be able to develop on.

4.7 Parsing system

To make sense of what is in the input .CSP file, the CSP++ framework needs to split it up and interpret it. This phase of the translation works in much the same way as a conventional compiler - it takes some textual representation as input, but instead of returning byte or machine code, it outputs the C++ representation. The next section will briefly discuss how such a system works, and we follow that by examining how it is implemented in CSP++.

4.7.1 Theory

To get some semantic meaning from a text, it must go through two phases of inspection, a lexical analysis and parsing. Lexical analysis amounts to splitting up the input into smaller chunks, which are known as tokens - this is just the mechanics of making the input text file readable. These tokens are then fed as input to a parser, which defines the meaning of each token with respect to some grammar.

More concretely, we have two common tools available to us, lex and yacc. To begin the process, a lexical analyser specification must be written. For us using lex, this amounts to writing a plain text .lex file. This file is based on regular expressions to match parts of the text, and when they match, fragments of C code can be run. To illustrate this, Listing 6 is a small specification, which will recognise any vowels or numeric characters.

To make this of any use, the commands shown in Listing 7 must be run. The lex tool generates the source code for a lexical analyser written in C. The gcc compiler is then used to compile this into a program, and the program is run. Once given an input string the lexical analyser reads the input string from start to finish, and wherever a match is found with the specification, the code associated with it is run. In this example that amounts to echoing the relevant piece of the input string, and telling the user what it is.

On a higher level than the lexical analyser is the parser. Once again we have a tool to assist us here, this time called yacc. The name yacc is actually an acronym, standing for ‘yet another compiler compiler’. During operation it calls the lexical analyser to provide it with tokens from the program text, and match them with a grammar specified in another input file. As yacc processes the input file, it will build up a tree representing the parsed data, where leaf nodes of the tree are terminal symbols, the same as those tokens returned by the lexical analysis. Yacc actually produces what is known as a left reduction parser. This means its grammar begins...
Listing 6: Example lexical analyser specification.

```c
/** Begin definition section */
%
#include <stdio.h>
#include "y.tab.h"
extern int yylval;
%

%option noyywrap
%
/** Begin rules section */
a | e | i | o | u | go { printf("%s is a vowel\n", yytext); }
[0-9]+ { printf("%s is a number\n", yytext); yylval = atoi(yytext); return NUMBER; }

return 0; /* End of token. */
. return yytext[0]; /* Return unrecognised tokens for yacc to deal with. */
```

Listing 7: Terminal output from lexical analysis.

daniel@EeePC:~/Desktop$ lex Vowels.lex
daniel@EeePC:~/Desktop$ gcc lex.yy.c
daniel@EeePC:~/Desktop$ ./a.out
daniel@EeePC:~/Desktop$ a: is a vowel
e: is a vowel
i: is a vowel
o: is a vowel
u: is a vowel
1234567890: is a number
with the largest object, such as a whole program, which is made up of many smaller objects, such as expressions and statements.

To make an example of such a parser, we will expand our lexical analyser above to become a simple calculator over integer input - recall that our previous analyser could recognise and return integer tokens. The specification file we wrote to allow yacc to generate our parser is `Calculator.y`, which is shown in Listing 8. The left reduction can be seen here, where the largest object is a statement, which in turn is made up of smaller expressions.

Once the yacc tool has been run on this file it can be compiled and linked with the lexical analyser, and run. When running, this version will evaluate expressions that are typed in by calling the lexical analyser, getting its tokens, and carrying out the associated routine. Listing 9 shows the commands issued to create the parser, and to evaluate an expression using it.

### 4.7.2 The CSP++ translator

Like our previous examples, the CSP++ lexical analyser and parser are built up using lex and yacc, so within the translator source directory we find the two files `expect`, `cspm.lex` and `cspm.y`. In fact, the open source equivalent of lex and yacc have been used, known as Flex and Bison respectively. This does not change any of the theory or implementation details. In the same sequence as we did our example, we first consider the lexical analyser specification file. It is heavily commented and includes some nice macros to allow more intuitive names for common regular expressions, such as those recognising alphanumeric characters and larger numbers.
All single character and comment tokens are defined, as well as some parts of CSP\textsubscript{M} that are not required in C++, such as channel and assert statements. This is because a channel variable means nothing in C++, and although assert statements are invaluable to tools such as FDR2, for this translation they can simply be matched, and thrown away. Valued tokens are defined, which have a little processing associated with them to help the search. These valued tokens will be important, because they pass data from the input file up to the parser, and the most common use for this is in declarations, for instance variables. There is no main method or user code in the \texttt{cspm.lex} file, as it is called by the yacc system where the code is kept.

The yacc file is significantly larger, and once again heavily commented. It sets some variables to control the flow of translation, including ways to identify fatal errors and handle them. The precedence of some rules are set, and then the CSP\textsubscript{M} grammar is laid out. It is here that the semantic meaning of the CSP\textsubscript{M} text file is defined, exactly as the semantic meaning of + and - were in our calculator example. A large section of code is given over to handling interrupts. We find the following familiar looking excerpt around two thirds of the way through the file:

```plaintext
exp : NUM { $$ = new PNnum( $1 ); } \\
    | ID { $$ = new PNvar( $1 ); } \\
    | TRUE { $$ = new PNlit( L_TRUE ); } \\
    | FALSE { $$ = new PNlit( L_FALSE ); } \\
    | exp '+' exp { $$ = new PNop( O_ADD, $1, $3 ); } \\
    | exp '-' exp { $$ = new PNop( O_SUB, $1, $3 ); } \\
    | exp '*' exp { $$ = new PNop( O_MUL, $1, $3 ); } \\
    | exp '/' exp { $$ = new PNop( O_DIV, $1, $3 ); } \\
    | exp '%' exp { $$ = new PNop( O_MOD, $1, $3 ); }
```

As one would expect, this defines the meaning of some literals, and then the behaviour of the familiar mathematical operators, so we can see this is a part of the code we may add to in our development. Every rule in the grammar will either instantiate a new parse node class, or add a parameter to what will later be a parse node.

The main method is the most interesting part of the file, as it can be regarded as translation headquarters - it is in control of the entire translation process. Yacc mandates that there must be two additional methods present, \texttt{yylex()} and \texttt{yyerror()}. Both are defined here. The \texttt{yylex()} method simply instantiates and makes a call to the Flex lexer, which will be used later to provide tokens. The \texttt{yyerror()} method is called whenever a problem has occurred in the parsing process, such as a syntax error. The fatal error flag is set true, and the line currently being read is printed to the standard error stream. This is what the user will see at translate time if they make a mistake writing their program.

The \texttt{main()} method begins by processing any command line flags that have been used when cspt was run at the command line. Table 2 summarises the flags that can be used with the translator. Next, input and output files are set up, and the output the user sees is printed to standard out: \texttt{Translating to file: ...}

The second stage of translation now begins, and a ‘scratch file’ is set up. This is used to store the code while it is being generated, ultimately this file will be removed and the finished code copied into the output file. The syntax tree is iterated, and the \texttt{gen()} method is called on each node in the tree. Some headers and include statements are printed into the file manually, and finally a hard coded main function is added. All of this is copied to the out file, and we are finished. This process is summarised and illustrated in Figure 12.

### 4.8 The parse node hierarchy

As seen in the forgoing section, the \texttt{ParseNode} objects are integral parts of the syntax tree that is built up from the lexical analysis and parsing phase. The \texttt{ParseNode} part of the system
Table 2: Command line flags: cspt <flags> <file.csp>

<table>
<thead>
<tr>
<th>Flag</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>-d</td>
<td>Debug mode. Prints the syntax tree being created to standard error.</td>
</tr>
<tr>
<td>-s</td>
<td>Source output. Will interleave translated C++ code with the CSP code that is associated with it. The CSP is commented out.</td>
</tr>
<tr>
<td>-t</td>
<td>Bison tracing. Will enable the debugging information from the Bison parser to be printed to standard out. This lists the tokens found, and how they are matched to the grammar rules</td>
</tr>
<tr>
<td>-w</td>
<td>Working directory. Files written will be saved to the current working directory.</td>
</tr>
</tbody>
</table>

Figure 12: The process of translation from CSP to C++.  

is an exemplary use of the object oriented paradigm, as full use is made of inheritance and encapsulation. This is illustrated in Figure 13.

The abstract class ParseNode is the top of this hierarchy, so we begin by looking here. It contains the variables and methods needed to store a node in the parse tree, and to print its code onto an output stream. The line number of its corresponding source is kept so that it can be reported along with errors, and indentation counters are kept for each output stream for use by pretty printers. The prep() and gen() methods are there for the code generation phase, and in this class are generic - though in practise are not often over ridden. The printing methods are used when the debug flag is used, so that the code can be output, usually to standard error.

The subclasses of ParseNode are specialised versions of their superclass. PNcop stands for Parse node (complex operator), and is used when an operator must maintain a list of arguments. This class has subclasses of its own, one for each type of complex operator that will need to be stored in the tree. PNcid is a Parse node (complex identifier). Once again this class can store a list of arguments along with its name, and perform its own code generation. It has its subclasses to take care of specialised complex identifiers.

The code generation phase itself amounts to walking this tree, by recursively calling the prep() and gen() methods. This will cause the C++ code to be written to the appropriate output stream, and end up in the output file.

Another important data structure is the symbol table - there are actually three such tables, one for agents, one for actions, and one for datums. In CSP a symbol is either a process name, a channel name or a variable name [Gar00]. They can be 'subscripted', that is given arguments, and if they are, these must be stored in the symbol tables too. The table is then read and modified by some parse nodes in the syntax tree, and will be used during the code generation phase.
Figure 13: Class hierarchy within ParseNode [Gar00].
Part II

New ideas, more technology
1 Beginning modifications

To begin modification of the code, we found it best to begin by creating a pair of exemplar files which describe the behaviour we are trying to create. The first type and operation we decided to implement was over sets, describing the empty set operator. The CSP file to demonstrate the behaviour of the empty set operator is given in Listing 10.

**Listing 10: Exemplar CSP file.**

```csp
channel correct, incorrect

SetA = {1, 2, 3}

SYS = if (empty(SetA) == false) then T else F

T = correct -> SKIP
F = incorrect -> SKIP
```

The next step was to identify what code needed to be generated in C++ to have equivalent semantic meaning. To ensure that we began with the same source code as all translated CSP, we modified an existing minimal example, translated by the CSP++ framework. Naturally, this breaks formalism but as a first prototype it proves invaluable as it serves as the benchmark code that we are aiming to generate. The code excerpt is shown in Listing 11. The changes are the declaration of a set, and of course the call to its empty method. In addition to this, not shown in the excerpt is a necessary include statement, without which the code will not compile. The behaviour of this small example was tested by compiling it along with the OOAF, and running it. As no functions outside of the C++ standard template libraries are used no changes to the OOAF are needed, but as complexity grows and new OOAF operations are needed the compilation step here will have to be omitted until extra code has been written.

**Listing 11: Exemplar C++ file.**

```cpp
std::set<int> SetA; // Declare the set we're using

AGENTPROC( SYS_ )

if (SetA.empty() == false) { // Check if the set is empty
    CHAIN0( T_ ); // If yes, invoke the "T" process (T = true)
}
else {
    CHAIN0( F_ ); // If not, invoke the "F" process (F = false)
}
```

Continuing from the analysis of CSP\textsubscript{M} types supported in the original version of CSP++ presented in Section 4.3, we have identified a possible C++ analogy for each operation. Wherever it is possible, we have drawn it from the C++ Standard Template Library (STL), as these are tried and tested algorithms we can rely on to operate correctly. Where algorithms aren’t available, or don’t quite suit our needs we will develop our own solution. Because this adds extra functionality which can be called on from the translated C++ code, we must include it as part of the OOAF. Table 3 shows the C++ analogies identified for each CSP operation.

Over the following subsections we will discuss the process of adding each data type, and the operators for each. In practice this was done iteratively, that is a single type was introduced, all of it’s operators added one by one, and then test cases were written for it. To make the discussion flow a little better we will break from this structure a little, and group by task in some cases.
Table 3: Support of CSP operators in the C++ STL.

<table>
<thead>
<tr>
<th>Data type</th>
<th>Operation</th>
<th>C++ analogy</th>
</tr>
</thead>
<tbody>
<tr>
<td>Integer</td>
<td>Multiplication</td>
<td>*</td>
</tr>
<tr>
<td>Integer</td>
<td>Unary minus</td>
<td>-</td>
</tr>
<tr>
<td>Integer</td>
<td>Addition</td>
<td>+</td>
</tr>
<tr>
<td>Integer</td>
<td>Subtraction</td>
<td>-</td>
</tr>
<tr>
<td>Integer</td>
<td>Division</td>
<td>/</td>
</tr>
<tr>
<td>Integer</td>
<td>Equality</td>
<td>==</td>
</tr>
<tr>
<td>Integer</td>
<td>&lt;, &gt;</td>
<td>&lt;, &gt;</td>
</tr>
<tr>
<td>Integer</td>
<td>≤, ≥</td>
<td>&lt;=, =&gt;</td>
</tr>
<tr>
<td>Integer</td>
<td>Modulo</td>
<td>%</td>
</tr>
<tr>
<td>Boolean</td>
<td>Equality</td>
<td>==</td>
</tr>
<tr>
<td>Boolean</td>
<td>And, Or</td>
<td>&amp; ,</td>
</tr>
<tr>
<td>Boolean</td>
<td>Not</td>
<td>!</td>
</tr>
<tr>
<td>Set</td>
<td>Union</td>
<td>STL algorithms : set_union()</td>
</tr>
<tr>
<td>Set</td>
<td>Intersection</td>
<td>STL algorithms : set_intersection()</td>
</tr>
<tr>
<td>Set</td>
<td>Difference</td>
<td>STL algorithms : set_difference()</td>
</tr>
<tr>
<td>Set</td>
<td>Distributed union</td>
<td>None</td>
</tr>
<tr>
<td>Set</td>
<td>Distributed intersection</td>
<td>None</td>
</tr>
<tr>
<td>Set</td>
<td>Member</td>
<td>Iterator find()</td>
</tr>
<tr>
<td>Set</td>
<td>Cardinality</td>
<td>.size()</td>
</tr>
<tr>
<td>Set</td>
<td>Empty</td>
<td>.empty()</td>
</tr>
<tr>
<td>Sequence</td>
<td>Length</td>
<td>List : size()</td>
</tr>
<tr>
<td>Sequence</td>
<td>Null</td>
<td>List : empty()</td>
</tr>
<tr>
<td>Sequence</td>
<td>Head</td>
<td>List : front()</td>
</tr>
<tr>
<td>Sequence</td>
<td>Tail</td>
<td>Iterative copy.</td>
</tr>
<tr>
<td>Sequence</td>
<td>Concatenation</td>
<td>List : merge()</td>
</tr>
<tr>
<td>Sequence</td>
<td>Element membership</td>
<td>Iterative comparison</td>
</tr>
<tr>
<td>Tuple</td>
<td>Declaration</td>
<td>Tuples not in STL - Boost maybe.</td>
</tr>
<tr>
<td>User defined</td>
<td>Definition of user defined function</td>
<td>Method.</td>
</tr>
</tbody>
</table>
1.1 Identification in the source code

Recall from our discussion on parsing theory, that the first step is to interpret the input file by using a lexical analyser. There is already such a lexer in CSP++, so we begin by modifying it. The first thing we must do is provide a way for the lexer to recognise when it has found an instance of a declaration of one of our data types. This is done using regular expressions, how each of them were defined is shown Listing 12.

Listing 12: Regular expressions to recognise sets, sequences and tuples.

<table>
<thead>
<tr>
<th>Type</th>
<th>Regular Expression</th>
</tr>
</thead>
<tbody>
<tr>
<td>setInt</td>
<td>&quot;{}</td>
</tr>
<tr>
<td>setBoolean</td>
<td>&quot;{}</td>
</tr>
<tr>
<td>sequenceInt</td>
<td>&quot;&lt;&gt;</td>
</tr>
<tr>
<td>sequenceBoolean</td>
<td>&quot;&lt;&gt;</td>
</tr>
<tr>
<td>nestedSetInt</td>
<td>&quot;(&quot;</td>
</tr>
<tr>
<td>tupleInt</td>
<td>&quot;(&quot;</td>
</tr>
</tbody>
</table>

The first observation is that there is some duplication, with only the data types being changed. It can be seen that this could be condensed quite easily: adding another line that states \texttt{type = num|boolean} and referring to \texttt{type} in the rules would suffice. However, this would mean we could only have one rule fire when these patterns are recognised, and we would have no type information to pass down into the code generation framework. By recognising each type separately we can use enumerated values later, which are then used to declare storage of the correct type in the C++ code. It does mean it’s a little less extendible, as for instance adding strings would mean adding another line here. With such a weakly typed algebra as CSP though, we can see no worthwhile alternative.

Each regular expression is organised in the same way. We start with the empty set, then a structure with just one element, then a structure with many elements. By adding a clause after the comma that matches zero or more white space characters, it does not matter how the user spaces the elements in their set. An exception to this structure is made where nesting is required. Here the first case allows a nested empty set, the second case allows many nested sets, each with one or more elements. Note that they can only be nested to one level, and they can only be of homogeneous type. The restriction on nesting could be lifted, but would enlarge the generated C++ code considerably, as each level would require an STL set, as well as the array needed to initialise it. The typing restriction is more difficult to lift. Each level of nesting is stored in one STL container, which usually can hold just one type. It could be declared more loosely, but then no type information would be retained so there would be no way to check for compatibility of operators at compile time. We feel that the work required to do this, and the potential for runtime errors it introduces is not outweighed by any potential advantage of being able to use heterogeneous data structures.

All that’s left to do in the lexer is to return some token to the parser. In addition to the token though, we must pass some data. The data we need is all of the matched declaration line, that is the set, all of its elements together with its identifier. Nothing is instantiated at this time, so it is just passed as a string value. To do this we write some more code, shown in Listing 13.

Listing 13: Returning valued tokens from the lexical analysis.

```c
{[id]} yyval.stringval = new string(yytext); return ID;
{[num]} yyval.intval = atoi(yytext); return NUM;
{[nestedSetInt]} yyval.stringval = new string(yytext); return NESTED_SET_INT;
{[setInt]} yyval.stringval = new string(yytext); return SET_INT;
{[setBoolean]} yyval.stringval = new string(yytext); return SET_BOOLEAN;
{[sequenceInt]} yyval.stringval = new string(yytext); return SEQUENCE_INT;
{[sequenceBoolean]} yyval.stringval = new string(yytext); return SEQUENCE_BOOLEAN;
{[tupleInt]} yyval.stringval = new string(yytext); return TUPLE_INT;
```
Having recognised a token in the input file, the parser needs to know what to do with it. This means it is time to modify the grammar given in the cspm.y file, by creating a rule that matches the whole declaration of a variable. In our case, a declaration looks loosely like this:

\[ ID = \text{StructureToken} \]

Lengthy examination of the existing grammar structure tells us that the place to put it is alongside the definition of a process, as this will not upset the priority of other rules when the ID = tokens are encountered. Before we can do that though, the tokens must be declared, along with what type of value they carry - all strings, in our case.

Each rule is added as another case of definition, as shown in Listing 14. Here we have omitted the actions that are carried out when a match is found, we will examine each rule in turn in the following subsections.

Listing 14: Rules recognising variable declarations.

<table>
<thead>
<tr>
<th>sigvar DEFINE SET_INT {}</th>
</tr>
</thead>
<tbody>
<tr>
<td>sigvar DEFINE SET_BOOL {}</td>
</tr>
<tr>
<td>sigvar DEFINE NESTED_SET_INT {}</td>
</tr>
<tr>
<td>sigvar DEFINE SEQUENCE_INT {}</td>
</tr>
<tr>
<td>sigvar DEFINE SEQUENCE_BOOL {}</td>
</tr>
<tr>
<td>sigvar DEFINE TUPLE_INT {}</td>
</tr>
</tbody>
</table>

The notion of a sigvar is previously defined in the parser, and can generate itself. It is a recursively defined structure allowing a simple identifier, or an identifier followed by some number of parameters, separated by commas and surrounded by parentheses.

1.2 Sets

Sets were the first data structure we introduced to the CSP++ framework. The regular expressions and lexer tokens used have already been described in the previous section, so we can begin here with an explanation of the process from the parser onwards.

Recall how the code generation system works. Every rule in the parser creates a new node in the parse tree, which when complete is traversed and each node has a gen() method called on it. In the spirit of this system, we must define a new parse node to hold the set, and give it a suitable code generation method to print itself to the output file.

We have defined our new parse node as PNsetDatum, a new subset of the PNtok class. The constructor code for it is brief, so given here in Listing 15.

Listing 15: Parse node class for set objects.

```
PNsetDatum::PNsetDatum(PNdefnID* i, string* value, SetType type)
{
    if(type == PNsetDatum::INTEGER) {
        // Add it to the datum table to do the declaration.
        datumTable.insert(make_pair( i->getID(),
            new SYdatum( i->getID() , 0, value , SYdatum::SET_INTEGER ) ));
    }
    else if (type == PNsetDatum::BOOLEAN) {
        // Add it to the datum table to do the declaration.
        datumTable.insert(make_pair( i->getID(),
            new SYdatum( i->getID() , 0, value , SYdatum::SET_BOOLEAN ) ));
    }
    // Add it to the constant table so it knows it can use it in operations.
    constantTable.insert(make_pair(i->getID(),
            new SYconstant(i->getID() , 0, *value )));
}
```

Note the parameters going into this method. The first parameter, the PNdefnID* i, is instantiated from the sigvar part of the parser rule, and holds all of the CSP code that was
to the left of the equality sign. The value parameter contains the value on the right hand side of the CSP definition, including the parentheses that surround it. The enumerated set type value is provided to allow a branching structure be used, in which different code is generated depending on the type being held in the structure.

An entry is made in the datum table to allow for generation, as this table is later iterated through, with each entry being called in turn. The entry into the constant table is important, as the framework will check this when a variable is used, and if the variable is not named there a translation error is raised letting the user know it has not been declared.

The generation method is interesting, and is included in the Symbols.cc file. In C++ one cannot instantiate a set directly with its elements, an array has to be used first which then has it's contents copied by the constructor of the set.

Listing 16: Generation code for set objects.

```cpp
if (type == SYdatum::SET_INTEGER) {
    // Print line declaring array used in the set constructor.
    os << "int *" << *plainName << "Array[] = " << *content << ";" << endl;
    // Print set declaration.
    os << "static std::set<int> " << *plainName << "( Array[] + CountElements(content) ) ; " << endl;
} else if (type == SYdatum::SET_BOOLEAN) {
    // Print line declaring array used in the set constructor.
    os << "bool *" << *plainName << "Array[] = " << *content << ";" << endl;
    // Print set declaration.
    os << "static std::set<bool> " << *plainName << "( Array[] + CountElements(content) ) ; " << endl;
}
```

The code in Listing 16 is what will write the C++ source to the output file, or out stream (os). This is where the set type information comes into play, declaring the array and set either to be integer or boolean. Obviously if more data types were to be added, more case distinctions would be placed here.

There is one more important thing to be done before the set can be used, which is to print the include statement. In the parser, we add a boolean variable to indicate whether or not set functionality has been used. When the rule fires, as well as instantiating the parse node we also set the boolean flag to true. In the parser main method, we add some code to print the `#include <set>` statement only when the flag is true.

At this stage, we have a system that can parse a declaration of a set in CSP\(_M\), and produce the corresponding declaration and instantiation in C++. To make this part of the exercise at all useful, we need to define the operators on our set, as at the moment they are simply unrecognised tokens. The first task is to define tokens for each operators keyword, and to get the lexer to recognise them:

- "empty" return EMPTY;
- "card" return LENGTH;
- "union" return UNION;
- "inter" return INTERSECTION;
- "diff" return DIFFERENCE;
• "member" return MEMBER;

The quoted left part of each item is the keyword, which we must match and should not change as it is defined by the CSP$_M$ language specification. The right hand part of the rule, in capitals, is the token we have defined to refer to it within the CSP++ system. Its value is arbitrary. Rules for each one of these tokens is defined in the yacc parser, and differ depending on the number of arguments. The rules for sets are shown in Listing 17.

Listing 17: Parser rules for set operators.

| $EMPTY \((\ 'exp' \ )\) \{ $\$ = new PNop( O EMPTY, 0, $3 ); \} |
| $LENGTH \((\ 'exp' \ )\) \{ $\$ = new PNop( O LENGTH, 0, $3 ); \} |
| $UNION \((\ 'exp', \ 'exp' \ )\) \{ $\$ = new PNop ( O UNION, $3, $5 ); \一字  
functionsUsed = true; \} |
| $INTERSECTION \((\ 'exp', \ 'exp' \ )\) \{ $\$ = new PNop ( O INTERSECTION, $3, $5 ); \一字  
functionsUsed = true; \} |
| $DIFFERENCE \((\ 'exp', \ 'exp' \ )\) \{ $\$ = new PNop ( O DIFFERENCE, $3, $5 ); \一字  
functionsUsed = true; \} |
| $MEMBER \((\ 'exp', \ 'exp' \ )\) \{ $\$ = new PNop ( O MEMBER, $3, $5 ); \} |

Once again, the | symbol denotes alternatives, and the capitals refer to the terminal symbols we defined previously. The dollar notation refers to the value of each part of the rule. Notice that for the union, intersection and difference operators we set the functions used flag to true. This is to allow an include statement to be set, which allows use of the functions we define in our own library, described later.

The code generation part is very simple, and built on top of the previous version of the CSP++ framework. A case distinction is made on the token type, distinguishing between ‘objectName.attribute()’ calls, ‘method(arg1, arg2)’ calls and ‘arg1 operator arg2’ calls.

The generation of code corresponding to the short hand CSP notation, for example \{1..3\}, is also supported.

1.2.1 Nested sets

Nested sets of booleans and integers are also supported, that is, sets of sets: SetA = \{ \{1, 2\}, \{3\}, {} \}. These are not so hard to generate, as the top level set can be regarded as a container for all of the subsets, which we can already generate as discussed previously.

New regular expressions are introduced in the lexer file, which simply identify a nested set as one or more sets surrounded by an outer set notation. The parser rule for them is more interesting, as it demonstrates the ancillary code required:

Listing 18: Parser rule for nested sets.

| sigvar DEFINE NESTED_SET_INT \{ numberOfNestedSets++; $\$ = new PNestedSetDatum((PNdefnID+)\$1, \new string(*$3), numberOfNestedSets, PNestedSetDatum::INTEGER); setUsed = true; nestedSetUsed = true; \} |

A counter of the number of nested sets used is maintained to allow them all to be instantiated correctly, as more code is relied upon to write a new method that populates the nested structures.
in C++. The pair of boolean flags are used to trigger the generation of this code, as well as the generation of include statements.

The new parse node, the nested set datum, contains a great deal of code dedicated to string processing, to chop up the declared set into each one of its sub-sets. For each sub-set a new C++ set is declared, the exact syntax of it being dependent on the type of the data being held. Once all the elements are done, the main set is declared, for instance in C++ as `std::set<
std::set<int> >`.

To add each sub-set to the main set, a method is written to the out file, with the set number included as part of its identifier. Finally, for each sub-set a call is made to instantiate a set datum parse node, which generates all the code required for them.

In the parser main method, the set number is used again, to add calls for each method that populates the main sets. This is placed among the first lines of the C++ main method, so that the set is ready for use when the processes are called.

### 1.3 Sequences

Sequences operate in a similar fashion to their set counterparts. To begin with, the notation of a sequence in CSP\textsubscript{M} is identical to a set with the exception of the `{...}` delimiters being switched to `<...>`. For that reason, the regular expressions in the lexer are very similar. The rules in the yacc parser are closely followed too, although this time another new parse node is instantiated, \texttt{PNsequenceDatum}. This node works on the same principals as the set node, adding the information to the two symbol tables.

This time, the following keywords and tokens are recognised:

- "empty" return EMPTY;
- "length" return LENGTH;
- "union" return UNION;
- "inter" return INTERSECTION;
- "member" return MEMBER;
- "null" return EMPTY;
- "head" return HEAD;
- "tail" return TAIL;
- "elem" return IS_ELEMENT;

The principal of their operation is identical to the set operators, so we do not repeat the discussion.

#### 1.3.1 Nested sequences

Nested sequences are supported, once again to contain elements of boolean or integer type. The architecture of this system is modelled on the nested set infrastructure, so we will not cover it in great detail here.

A key difference is that the `<...>` notation of CSP\textsubscript{M} is not used in C++, so amongst the string processing code we must add lines that replace these with standard curly braces, `{...}`. A counter is used to maintain methods to instantiate the main top level set, and lines are added to the C++ main method to call each one of them.
1.4 CSP$_M$ functions

To support the operations that are defined over the new data types, we have extended the OOAF by two new files, CSPmFunctions.h and CSPmFunctions.cc. These exist to provide functionality that is not present within the Standard Template Library of C++, or where more compact syntax is convenient for our code generation. Inclusion as part of the OOAF was not difficult to achieve, as the libcspxx.a archive is built from a top level make file. Editing this allows us to include the new files, which when compiled along with the translated code with its include statements has linking carried out as usual by the C++ compiler.

The following functions are implemented:

- set<int> SetUnion (set<int> A, set<int> B);
- set<bool> SetUnion (set<bool> A, set<bool> B);
- set<int> SetIntersection (set<int> A, set<int> B);
- set<bool> SetIntersection (set<bool> A, set<bool> B);
- set<int> SetDifference (set<int> A, set<int> B);
- set<bool> SetDifference (set<bool> A, set<bool> B);
- set<int> DistributedUnion (set< set<int> > A);
- set<bool> DistributedUnion (set< set<bool> > A);
- set<int> DistributedIntersection (set< set<int> > A);
- set<bool> DistributedIntersection (set< set<bool> > A);
- list<int> ListTail (list<int> A);
- list<bool> ListTail (list<bool> A);
- bool SetMember(int A, set<int> B);
- bool SetMember(bool A, set<bool> B);
- bool ListMember(int A, list<int> B);
- bool ListMember(bool A, list<bool> B);
- list<int> ListConcatenation(list< list<int> > A);
- list<bool> ListConcatenation(list< list<bool> > A);

The union, intersection and difference functions are used here to provide a more convenient wrapper to the STL functions that make them work. An example of this is the set difference method, given below:

Listing 19: Set difference method.

```c++
set<int> CSPmFunctions::SetDifference (set<int> A, set<int> B) {
    set<int> newSet;
    set_difference( A.begin(), A.end(), B.begin(), B.end(),
                   inserter_iterator < set<int> >( newSet, newSet.begin() ) );
    return newSet;
}
```
Instead of having to use the standard set_difference syntax, with all the references to elements in the set, it is easier to generate code that simply calls the function with two sets, returning a new set as the result. The result of this is simpler, easier to read code, which is more reliable than the complex alternative. It will also reduce the number of lines generated in the C++ output files, as the OOAF library code is transparent to the user.

Other methods are more interesting as they require a little more processing, so we include a few lines on each here.

- **CSPmFunctions::DistributedUnion**: This is an operation over nested structures, which will return a non-nested structure of the same type. Our implementation of this is to create a new structure, in which every element of the nested one is inserted. The structures themselves do not allow duplicate elements, so no protection is required to prevent repeat insertion.

- **CSPmFunctions::DistributedIntersection**: This function is more complex than a distributed union, as the structure it returns will contain only those elements present in all substructures of the nested one. We do this by starting with the first substructure, and iterating through it. For each element in it we test for its membership of all remaining substructures. As soon as it is not found in one of them, the search is stopped in the interests of efficiency. If the element is found in all substructures, only then will it be added to the output, and the search continues.

  This process could be made more efficient by identifying the smallest substructure, and using its elements as the basis for the search. By doing so it is guaranteed that the minimum number of search steps are needed, as there are the fewest number of element comparisons to be made. We have chosen not to implement this, as there would be an over head of iterating through the main structure to find the smallest substructure. On large structures this would undoubtedly pay off, however taking into account the nature of CSP specifications we feel it does not warrant the time or complexity.

- **CSPmFunctions::ListTail**: A tail function returns all but the first element of a list. We implement this by adding all of the elements but the first to a new list, and returning it. Where the empty list is passed to this method, a new empty list is returned in its place.

- **CSPmFunctions::SetMember**: This uses the find() method of the STL, returning true where the element given was found, and false in all other cases. List member works in a similar way, but has to perform iterative comparison as the STL has no analogy for find() defined over lists.

- **CSPmFunctions::ListConcatenation**: This takes advantage of the STL merge() operation, by iteratively applying it to each sub-list, and returning the result.

Although it forms part of the OOAF, this code can be compiled and used in a stand-alone situation. When we developed we carried out testing in this way, but it is tested further as part of the CSP++ system in the CPPUnit test cases.

### 1.5 User defined functions

User defined functions are of key importance in CSP, as they allow the user to carry out some operation over some data. Throughout this discussion we will need an example, so the minimal example of a function that increments an integer is a good choice:

\[
\text{inc}(n) = n + 1
\]
CSP is an example of a functional language, so variables and side affects cannot be used. Concretely, this means that the following script is disallowed:

```plaintext
SomeVariable = 1
inc() = SomeVariable + 1
```

Although the pure notion of a function need not have parameters, we will exclude those without from our implementation. This is because a function with no parameter is nothing more than a constant, for instance `func = 1`. Both syntactically and semantically this is indistinguishable from the CSP concept of a variable, so there is no sense in duplicating what is done.

In general, the structure we are identifying is an ID followed by one or more parameters, an equality sign, and typically some equation on the right hand side defining the operation to be carried out. This equation is further complicated by the inclusion of branching statements, which can themselves be nested.

The clear C++ analogy here is a method, carrying the same ID and parameters as the CSP function. For our increment example:

```c++
int inc(int n) {
    return n + 1;
}
```

Note the differences of inclusion of types, return statements and statement termination symbols.

To begin adding this kind of structure we don’t need any new tokens, so for now the lexer remains unchanged. Consulting the parser grammar introduces a problem: the sequence of ID, equality and expression we have identified is identical to that of a process, but must be handled differently. To overcome this, we have introduced the CSP++ preprocessor.

### 1.6 CSP++ preprocessor

Preprocessors are not unusual, particularly in safety critical methods. C++ has one, which handles statements such as the `#include` and `#define` lines. The safety critical language SPARK Ada has provided the inspiration for our CSP++ preprocessor, which works by recognising some symbols in the code, and carrying out processing as directed by them.

SPARK Ada is a subset of the larger language Ada, which has been defined such that it does not violate any properties often required in safety critical systems, such as space and time bounds. To enforce these rules a set of preprocessor directives were created, such as `--# post X = X + 1;`, which states that the value of X after running the method should be the original value of X, with the addition of one. The important point to note here is that in Ada - not the subset, SPARK Ada - the character sequence `--` is recognised as a comment. Thus, to compile a SPARK Ada system the author first runs the Examiner tool, which recognises the lines prefixed with `--#`, and ensures that the safety critical constraints are met. Once this stage is complete, a standard Ada compiler is used to produce the executable, which raises no syntax errors as the preprocessor directives are ignored as comments.

We use a very similar approach to modify the structure of the CSP input files to the translator. To allow a function to be recognised in the grammar and distinguished from a process, we will prefix it with some new token. However, if this was done in the pure CSP source code, it would no longer be syntactically valid, and we would not be able to use it with any other tools that recognised CSP. Primarily these tools are verification aids, without the use of which our development paradigm would fall apart. To overcome this we use the same approach as SPARK Ada. A CSP comment is prefixed with `--`, exactly like an Ada comment. We will define a preprocessor symbol, `--#Function`, which the author will include in the CSP input file on the
line immediately preceding the function definition. Thus, our increment example now looks like this:

```csp
−−#Function
inc(n) = n + 1
```

The purpose of the preprocessor is to transform this line into a function definition containing the extra token that distinguishes the function from a process. After running, it will produce a pseudo-CSP file that looks like this:

```csp
#Function inc(n) = n + 1
```

There are two things to note here. First, the blank line that used to hold the preprocessor directive is preserved - this is so that in the event of a syntax error line numbers that are reported match the original input file. The second point is that the comment sequence has disappeared. This is of course not valid CSP

M, but is recognised by our parser.

This output is written into a new file, which has the name of the original input file prefixed by “Preprocessed-”. Where translation is successful this file is automatically removed, if some error occurs it is not deleted, allowing the user to investigate this step too. The implementation of the preprocessor is fairly simple, given a file name as input it will open it, and scan the file for preprocessor directives. Where a line is encountered with no directive it is copied to the output file, where one is found a blank line is inserted, the #Function tag put in place, and the line appended to it.

### 1.7 Handling the preprocessed file

Now that our preprocessor allows us to recognise function declarations in the CSP code, there are two distinct next steps. The first is to translate the function definition and body into C++, the second is to allow this code to be called, and the result used.

The first step is the single change to the lexer, which adds the tag #Function as a simple token, so that it can be used in the parser. We have already identified that the definition of a function is closely analogous to that of a process, so it is logical that the rules be added in the same place. Fortunately, the modular form of the grammar means some of the work is done for us, as we can reuse the notion of an expression to form the the right hand side of the function body. Thus, the parsing rule looks like this:

```csp
FUNCTION sigvar DEFINE exp { $$ = new PNfunction((PNdefnID)$2 , $4) ; };
```

It is one of the alternatives listed under the heading of defn, which makes sense as it is one of the forms of definition available in CSP

M. The function token is recognised first, which is the one put in place by the preprocessor and distinguishing it from other definitions. The sigvar, as previously discussed, contains all the data for the identifier and any parameters, followed by the equality sign, and then some expression. The expression is already defined over all mathematical operations, including those involving variables, but functionality for branching must be added as new alternatives:

```csp
| IF exp THEN exp { $$ = new PNifthen( $2 , $4 ) ; } |
| IF exp THEN exp ELSE exp
| { $$ = new PNelse( new PNifthen( $2 , $4 ) , $6 ) ; } |
```

Although these rules reuse the same parse nodes as the previous definition of branching, they are distinct from the IF exp THEN agent ELSE agent definition by the use of further expressions, instead of process definitions.
This puts in place enough to allow a function to be parsed, but to generate code for it, detail is needed for the new parse node, PNfunction. The constructor just performs assignment to local variables, so all of the logic is encapsulated within the code generation method.

Generation begins by interrogating the PNdefnID for the name of the function, then if it has any parameters, printing those with associated commas and parentheses. To generate the signature we assume that only integer data types are to be used as parameters, and that all functions return an integer value. We feel that neither of these restrictions impose unduly on the variety of specifications that can be translated, though this could be expanded in a future implementation in fairly straightforward ways - such as passing type information within the PNdefnID.

The rest of the generation process relies heavily on the notion of a ‘list of parse node pointers’, or LOPNP. This is a type defined as part of the CSP++ framework, that is used as a concrete representation of the parse tree. Considering our increment function in CSP\(_M\), the parse tree has an internal node for the addition operator, and a leaf for each operand.

The generation of code involving branching gives rise to some new problems, particularly in placing return statements. In general, something like our increment function has just one return statement, at the beginning of the line that defines its equation. This is trivial to generate, as one simply outputs the keyword return, then outputs the code for the equation. As soon as branching is introduced, there must be more than one path of execution through the function, and therefore more than one exit point. Consider the following in CSP\(_M\):

\[
\text{incIfOdd}(n) = \text{if } (n \% 2 == 1) \text{ then } n + 1 \text{ else } n
\]

is semantically equivalent in C++ as

```cpp
int incIfOdd(int n) {
  if (n % 2 == 1) {
    return n + 1;
  } else {
    return n;
  }
}
```

Note the presence of the two return statements that need to be added within the expression. To allow the generation of branching statements in the context of a function we need to somehow change the way they are generated. To do this, we have written two methods that carry out traversal of the parse tree. The first exploits a method in the ParseNode superclass, which returns whether or not a parse node in the tree is a branching statement. This is simply a method that returns false, which is overridden in PNifThen and in PNelse to return true.

To set the context of a branching node, they contain a local boolean variable indicating whether or not they are part of a function. Within the function node generation method, the parse tree is iterated, and if a branching node is encountered, the context flag is set to true. A flag local to the function generation method is also maintained, which will suppress generation of single return statements in the presence of branching.

The generation methods of those nodes responsible for the branching is modified to take account of the context flag. Whenever it is set true, that is to indicate the node is part of a function, the additional return statements are generated ahead of each expression.

All of the above steps are sufficient to generate the function definition and body within the source code, which is syntactically valid in C++, and equivalent to the CSP that it is translated from. This is, of course, useless without code that allows the function to be called, and the return value be used.

There are several valid ways in which a function can be called in CSP\(_M\), but at this stage we have implemented just one. This is to call a function as a parameter to a new process, such
as Proc(inc(n)), which was chosen as it has direct relevance to the case study examples we wish to translate. An example of another function call is on an output statement, for instance chan!inc(n). This has little practical purpose beyond debugging, though the addition of it would not be too challenging as most code could be directly reused.

The reusable part of the code is a new non-terminal introduced to the grammar, shown below:

```
functionCall:
  ID '(' exps ')' { $$ = new PNfunctionCall($1, $3); }
  | ID '(' varList ')' { $$ = new PNfunctionCall($1, $3); }
```

The first rule matches an identifier, such as our example inc, the opening bracket, some expression, and the closing bracket. The expressions here can be anything from a single variable, to a mathematical expression, allowing something like inc(n * 2) be called.

The reference to a varList is another non-terminal, which allows a recursively defined list in which other functions may be called. This means that given a second function, something like inc(mult(n, sub(m, o))) can be used. The rules for a varList are as follows:

```
varList:
  ID '(' numvars ')' { $$ = new LOPNP( 1, new PNfunctionCall($1, $3) ); }
  | varList ',' ID '(' numvars ')' { $1 -> push_back(new PNfunctionCall($3, $5)); $$ = $1; }
```

In the first case, the simple identifier followed by terminal symbols, the parameters, are instantiated in a new function call parse node, and placed in a list of parse node pointers. The second case, more complex, is when a further varList is followed by an identifier and parameters. Here, the concrete part is instantiated into a parse node, which is pushed back into the list, allowing processing to continue in the recursive call.

The code generation of function calls is entirely held in the function call parse node. Regardless of the complexity of the call the first step is to print the identifier of the function, followed by an opening bracket - so far, inc(). Next, the list of parse node pointers is iterated, recursively calling the generation method of each node. This generates each of the parameters, so in all cases but the first a comma is generated to separate the elements of the list. Finally, once the end of the list has been reached, the closing bracket is printed, giving us inc(n).

2 CPPUnit testing

CPPUnit is one of the so called xUnit family of tools. It is a framework that provides unit testing functionality to C++ in a fairly simple and straightforward way. The family of tools started with SUnit, a framework for testing Smalltalk programs in 1999, and has since been ported to many other languages such as JUnit for Java, the most widely used version, PyUnit for Python, and vbUnit for Visual Basic, to name but a few [Ham04].

Figure 14 shows the basic architecture of an xUnit system. The fundamental class is Test, which is extended by every test method the author implements. The other methods either serve the framework, such as TestRunner, or allow more advanced functionality, such as TestFixture, which can be used to instantiate objects required in test cases, and therefore keeps the repetition of code to a minimum. The TestSuite class is a very useful class, as it allows the author to group together tests as required, for instance all tests on a certain area of functionality. Finally, once tests have been run, the framework will usually produce a textual report of what has been achieved. In some cases this can be generated as an HTML report, detailing what tests were carried out and with information on pass or fail status, which can easily be included into a report.
We didn’t select CPPUnit as the framework for this project, the existing team had already implemented many tests using it. The testing methodology employed here is a little unusual though, and to a certain extent driven more by the make files than the CPPUnit framework itself. The reason for this is that usually the system under test is represented as C++ code, which is compiled along with the CPPUnit libraries, and executed to obtain a test verdict. While the compilation stage of our testing is unchanged, the input to the tests is CSP code. This means every CSP file must be translated, linked with any user coded functions if required, and only then compiled and executed with CPPUnit. This means that effectively the testing can be run in three phases. In the first phase, driven by the make files, all of the CSP source files are translated by the cspt tool. This could lead to failures because of a problem within the translator, where a syntax error is incorrectly raised, which will cause the testing process to stop. The second phase is the compilation of the translated C++ code with the standard GNU compiler. Every test file is compiled, and if any non-legal C++ code was generated the compiler will raise an error, and once again the testing process would be halted. Only in the third and final phase is any of the code run, and assertions made about the outcome of running the code checked. Here any runtime errors can be caught, but more importantly so can any deviation from the intended semantic meaning.

As a slightly philosophical note, the testing phase of this project is arguably the most important part of it. We are aiming to create a tool for formal methods, so some guarantee of its correctness is vital, which could be provided through testing, validation or verification. Although formal techniques could be applied to prove the semantic equivalent of constructs between CSP and C++, it goes beyond the scope of this project - but could be completed as part of future work. Instead, the congruence of each construct is matched in at least one test case, and as the test cases become more complex, constructs are reused to allow new ones to be used. An example of this is the initial testing of an if . . . then statement, and later the testing of a set union operator, where the outcome of the test depends on the evaluation of such a branch. Whilst at first the reliance upon a system under test to provide results about itself may seem unwise, the tests are executed in a linear fashion, so the correctness of the basic parts
is shown before they are used in the wider context.

We have extended the existing forty CSP files of test scripts with an additional four, which exercise the functionality we have introduced over sets and sequences, both for integer and boolean operators. The scripts are set out to exercise each operator defined on a data type, testing the next one only when its predecessor has passed. In the event of a failure simulating the test file with ProBe and selectively translating it will pinpoint the statement that causes the problem. We have included the code used in Appendix D, and Figure 15 shows the results of running the entire test suite.

![Figure 15: Terminal outcome of running the test suite.](image)

## 3 Demonstration

This section of the thesis is dedicated to evidencing the work we have done, as well as illustrating the development paradigm we aim to promote. We will consider three examples, each with differing areas of focus, and different levels of verification. The first is a small mathematical puzzle, which demonstrates the application of formal methods to provide an answer to something not immediately obvious. The implementation of this is not so useful, but confirms what is shown during the model checking phase. The second is an example of linear sorting, after simulating this we provide an executable implementation of the sorting algorithm. Finally we have the “concrete coffin” example, instigated by our industrial sponsor Atego. It demonstrates the development from UML diagrams through CSP to an executable system that could be used in the field to determine the safety of an escapology stunt.
3.1 Children’s puzzle

The children’s puzzle is an example of a mathematical curiosity, which cannot be traced back to any single reliable source. It is a scenario that demonstrates the impact of individual synchronised activities on a larger global system.

The scenario is set up such that there are some \( n \) children sat in a class room, such that \( n \geq 2 \). The children sit side by side with each other, forming a circle. Each child possesses some even number of sweets, and the teacher in the room holds a - possibly infinite - supply of additional sweets.

Once the system is arranged, that is our children are sat on the floor in a circle and provided with some arbitrary initial number of sweets, they begin carrying out a sequence of actions. Any child may begin, by passing half of their sweets to the child on their left. If this leaves the child with an odd number of sweets, they receive a single additional sweet from the teacher to make up the number they hold to an even number. This process then continues with the next child, handing on half of the sweets they possess, and receiving an additional sweet if required.

There are some obvious questions to be asked here:

- Will the teacher require an infinite supply of addition sweets: that is, will all children eventually always hold an even number of sweets without requiring topping up?
- Will the number of sweets in the system of children eventually stabilise: that is, will all children hold an equal number of sweets, and not receive any additional ones?

Both of these questions can be answered, and indeed proved, arithmetically. We do not provide this proof here as it does not add value to our study of the problem, but further description can be found in [CRS12].

For our purposes, we will model the system in CSP, and carry out some model checking on it using the tool FDR2. Once this model checking is complete we will pass the script through the CSP++ translator and execute the code created, to observe each of our children in action.

The CSP code to model the puzzle is given in Listing 20. Considering the script from the top, four channels are put in place. Each of the \( c \) channels represents the actions each of three children can carry out, passing or receiving sweets between each other. The \( d \) channel has been introduced simply for visibility during execution. As the actions the children carry out are not visible, they are output on this channel instead so we can keep track of how many sweets each child possesses at each step.

The role of the teacher is modelled by the \texttt{fill(n)} function. Note the preprocessor tag that precedes the definition of the function, interpreted in CSP as a comment. If passed an odd value for \( n \) the function will return \( n + 1 \), the sweet the child requires to top themselves up to an even number.

Then follows the three child processes. Each one can send half of their sweets to the next child, show that action on the output channel, receive some sweets, and collect the additional sweet if required before returning to their original state. The \texttt{SYS} process composes the three children in parallel, allowing them to communicate with each other by the channels discussed above.

To check whether or not the system will stabilise, and therefore answer the questions posed above, [CRS12] introduced the \texttt{StableAfter} process. This formalises the notion of stabilising within a certain number of iterations, and is model checked for increasing numbers of iterations by trace refinement in the assert statements at the bottom of the file.

Before translating the code, we use the tool FDR2 to evaluate the assert statements, which automatically carries out our model checking. The result of this is shown in Figure 16, there a red cross indicates a failed assertion, and a green tick a successful one.
Listing 20: CSP code modelling the children’s puzzle.

channel c0 : {0..4}
channel c1 : {0..4}
channel c2 : {0..4}
channel d : {0..2}.{0..4}
channel action

—#Function
fill(n) = if (n % 2 == 0) then n else n + 1

Child(0,x) = (c1!x/2 -> d.1.x/2 -> c0?y -> Child(0, fill((x/2) + y)))
          [,] (c0?y -> c1!x/2 -> d.1.x/2 -> Child(0, fill((x/2) + y)))

Child(1,x) = (c2!x/2 -> d.2.x/2 -> c1?y -> Child(1, fill((x/2) + y)))
          [,] (c1?y -> c2!x/2 -> d.2.x/2 -> Child(1, fill((x/2) + y)))

Child(2,x) = (c0!x/2 -> d.0.x/2 -> c2?y -> Child(2, fill((x/2) + y)))
          [,] (c2?y -> c0!x/2 -> d.0.x/2 -> Child(2, fill((x/2) + y)))

StableAfter(n) = if (n > 0) then c0?x -> StableAfter(n - 1)
          [,] c1?x -> StableAfter(n - 1)
          [,] c2?x -> StableAfter(n - 1)
else Stable

Stable = c0!2 -> Stable [,] c1!2 -> Stable [,] c2!2 -> Stable

SYS = (Child(0,0) [,][{c1}][{c0,c2}]) Child(1,2) [,][{c0,c2}][{c0,c2}]) Child(2,4)

assert StableAfter(3) [T= SYS \ {d}]
assert StableAfter(6) [T= SYS \ {d}]
assert StableAfter(9) [T= SYS \ {d}]
assert StableAfter(12) [T= SYS \ {d}]
assert SYS : [deterministic]
assert SYS : [livelock free]
assert SYS : [deadlock free]
Figure 16: Model checking the children’s puzzle in FDR2.

Note that in the assert statement all actions on channel $d$ are hidden. This is because it is purely an output action, so is not necessary in the model of stability. Now that we have proven the properties we need for the system, we can put it through our translator, compile the output C++ code, and run it to observe what happens.

Figure 17: Children’s puzzle running at the terminal.

Figure 17 shows the terminal output that occurs when the system is run. The first parameter on the $d$ channel is the identifying number of the child, the second parameter is the number of sweets that it holds at that time. We can see that after eight iterations, the system stabilizes with every child holding two sweets - just as we verified it would using FDR2. The execution will continue to run infinitely until we terminate it, as no end criteria was specified in the CSP.

45
code. No further steps happen, the children just infinitely exchange sweets, always having two in their possession. For the interested reader, the generated C++ code is given in Appendix E.

### 3.2 Linear sorting algorithm

The linear sorting algorithm is a primitive but provable algorithm to sort a list of integer values. This is a more academic example than the previous one, and will demonstrate different aspects of the tool and paradigm. This time the focus is not on verification, although it is possible, but on the concurrency being used between each sorting cell, and on the linking of user coded functions to the formally developed sorting code.

Before examining the translation of the algorithm, we will first take a high level look at how it works. Figure 18 shows an example run, sorting the a stream of binary integers, 0 1 1 0. Each step in the sorting algorithm is shown progressing down the diagram, with input streams on the left, and the output stream on the right.

The basic principle of operation is that each bit is read into the left most sorting cell. If it is empty, the value read in is stored there. If it is non-empty, the new value is compared with the stored one. The larger one is moved to a cell on the right - and compared with its contents, if present - and the smaller one stored in its place. In our diagram, the first step reads in a zero, which is stored in the first cell. Step two reads in a one, which is compared to the zero, found to be larger, and stored in the next cell to the right. The third step reads in a second one, which is compared to the zero, found to be larger, and then stored alongside the first one value. The fourth step reads in the final zero, which is compared to the zero stored in the first cell, and the other values are moved down in order that it be stored in its place. As all cells in the array are now filled, the final step flushes them to the output stream, this time as a sorted list. The CSP code that implements this behaviour is shown in Listing 21.

Sorting a stream consisting of only zeros and ones is not so interesting, but discussed and proven in [Dav12] is the 0-1 principle. This states that if every 0-1-sequence is sorted by some network $N$, then every arbitrary sequence is sorted by $N$. This means that we can extend our implementation to sort an arbitrary array of integers.

Before we translate the CSP code, we simulate it in the tool ProBe, to check it is sorting correctly. This is shown in Figure 19, where the sorted output of 0 1 0 0 is shown.

This can be translated in CSP++ with no changes. Compiling and running this will result in a working system, that calls for input on channel values, and ultimately outputs the sorted values. We wish to improve on this initial result by linking user coded functions that allow us
Listing 21: CSP code modelling the linear sorting array.

channel c0 : {0, 1}
channel c1 : {0, 1}
channel c2 : {0, 1}
channel c3 : {0, 1}
channel out0 : {0, 1}
channel out1 : {0, 1}
channel out2 : {0, 1}
channel out3 : {0, 1}

P0 = c0?x -> PM0(x, 1)
PM0(s, k) = if (k == 4) then
    out0!s -> SKIP else c0?x -> if (x > s)
    then c1!x -> PM0(s, k+1) else c1!s -> PM0(x, k+1)

P1 = c1?x -> PM1(x, 1)
PM1(s, k) = if (k == 3) then
    out1!s -> SKIP else c1?x -> if (x > s)
    then c2!x -> PM1(s, k+1) else c2!s -> PM1(x, k+1)

P2 = c2?x -> PM2(x, 1)
PM2(s, k) = if (k == 2) then
    out2!s -> SKIP else c2?x -> if (x > s)
    then c3!x -> PM2(s, k+1) else c3!s -> PM2(x, k+1)

P3 = c3?x -> PM3(x, 1)
PM3(s, k) = out3!s -> SKIP

SYS = ((P0 || { c1 } || P1) || { c2 } || P2) || { c3 } || P3
to read in values to a data structure, sort them, and place them back into another structure. Because of the concurrency the output from the cells is not guaranteed in any order, so one has to check the cell identifier to check that the output is sorted - we aim to present this better too.

To do this, we write a new file in C++, with one method for each action we intend to link with. Each method has a set signature that we must use, for example in Listing 22.

Listing 22: Example user coded function method.

```c++
void output0 ( ActionType a,
              ActionRef * r,
              Var * v,
              Lit * l ) {

  output[0] = *l;
  done1 = true;
  checkFinished();
}
```

We write a method to create a C++ standard list object, and code to prompt the user to place four integer values into it from the terminal. This is to be linked to the $c_0$ channel in the CSP code, where the first values are read into the sorting cells. We also write other methods linked to each output channel, that store the output value into the respective position in a C++ array. Once all of the output is done, a final method prints the output to the screen in order, by iterating the array. To actually link the events and methods, one first compiles the user coded functions to a C++ object file, and then uses the g++ compilers -D option, to define macros that allow the link. Full source code for both the translated sorting algorithm, and the user coded functions that are linked with it are given in Appendix F. The process of linking and running the sorting system is shown in Figure 20.
3.3 Escapology: The concrete coffin

The purpose of this oddly named example is to demonstrate the development paradigm we are advocating, from end to end. We will begin by examining a parametric UML diagram provided by our industrial partner, used in their book [HP11]. From this we will extract some CSP code, carry out verification to ensure some safety properties are met, and then translate the code using CSP++ to create an executable solution that will tell the user whether or not it is safe to go ahead with a certain stunt.

The so called concrete coffin is a stunt in escapology, where an individual is placed within a coffin, which is lowered into a hole, and covered with some material, typically concrete. The task of the escapist is to get free from the coffin and pass through the material covering it, unharmed. Besides the obvious difficulty, there are several factors that make this stunt particularly dangerous:

![Figure 20: Linear sorting array with user coded functions running in the terminal.](image)

![Figure 21: The concrete coffin stunt.](image)
• Coffin: the strength of the wood used to build the coffin plays a factor, because it must be strong enough to withstand the load of the material above it, without collapsing. The size of the coffin also determines how much air can be held within it, and therefore the amount of oxygen available to our escapist.

• Fill time: The material covering the coffin has to be put in place after the coffin has been lowered in to the hole, which takes some time out of the oxygen supply for the escapist. Depending on the power of the pump or other mechanism being used to put the material in place the amount of oxygen used will vary.

• Hole size: The volume of the hole that the coffin is lowered into can be critical, particularly the depth of it. The deeper the hole, and the greater the density of the material being used to cover the coffin, the greater the force the coffin must withstand - and of course the greater the distance the escapist must travel.

• Escapist: The fitness of the individual inside the coffin is important, because the fitter he is the less oxygen he will need to consume when exerting himself to escape to the surface. The physical size - volume - of the escapist plays a part too, as the larger he is the more of the space inside the coffin he takes up, which as we’ve already noted is related to the oxygen supply.

• Rescue: If the stunt becomes unlikely to succeed, we must put something in place to rescue our escapist. This will take a certain amount of time depending on what mechanism we put in place, be it a second pump, or a man with a shovel. The duration taken to safely extract the escapist must be subtracted from the maximum length of time he can survive in the coffin, the result becoming the latest point at which the safety team can intervene.

Figure 22: UML diagram describing parametric constraints on the concrete coffin stunt.
Figure 22 shows the parametric constraint UML diagram for the safety aspect of the stunt. Parameters are shown in the plain rectangles, functions and decision are in the large soft-edged squares. As discussed in the introduction section on the UML, techniques and tools exist to transform such a diagram into CSP. Because that transformation is not the subject of this project, we do not go into any detail of this stage, and in fact the transformation was done manually in this case.

In CSP we have modelled each parameter as an integer variable. The functions shown in UML are analogous to those used in CSP, so we translated them directly. The decisions are a little more complex, and allow us to exploit the concurrency capabilities of CSP by carrying out each part of the decision independently, but coming together in a common decision.

Listing 23: CSP code modelling the concrete coffin problem.

```csp
channel goAhead, endStunt, action
channel breathDecision, equipmentDecision : Bool

CoffinHoleLength = 10
CoffinHoleWidth = 4
CoffinHoleHeight = 4
CoffinHeight = 3
CoffinWidth = 2
CoffinLength = 7
CoffinCrushPressure = 200
Gravity = 10
FluidDensity = 10
EscapePumpRate = 20
EscapologistBMax = 2300

—#Function
Volume(lengthParam, width, height) = lengthParam * width * height
—#Function
Mass(density, gravity) = density * gravity
—#Function
Force(mass, acceleration) = mass * acceleration
—#Function
SurfaceArea(width, lengthParam) = width * lengthParam
—#Function
Pressure(force, area) = force / area
—#Function
FillTime(volume, pumpRate) = volume * pumpRate

Decision1 = if (EscapologistBMax > FillTime(Volume(CoffinHeight, CoffinWidth, CoffinLength), EscapePumpRate)) then BreathGo else BreathEnd
BreathGo = breathDecision!true -> SKIP
BreathEnd = breathDecision!false -> SKIP

Decision2 = if (Pressure(Force(Mass(FluidDensity, Gravity), Gravity), Gravity) < CoffinCrushPressure) then EquipmentGo else EquipmentEnd
EquipmentGo = equipmentDecision!true -> SKIP
EquipmentEnd = equipmentDecision!false -> SKIP

Go = goAhead -> SKIP
End = endStunt -> SKIP

Decision = (Decision1 ||| Decision2)
```

51
Combined = breathDecision?bd -> equipmentDecision?ed -> Overall(bd, ed) 
   \[ \]
   \[ \]
equipmentDecision?ed -> breathDecision?bd -> Overall(bd, ed) 

Overall(bd, ed) = if (bd and ed) then Go else End 

SYS = Decision 

Liveness = goAhead -> SKIP |\~| endStunt -> SKIP 

Deadlock = action -> Deadlock 

assert SYS :[liveLock free] 
assert SYS :[deterministic] 
assert Liveness [F= Decision \ \{ breathDecision , equipmentDecision \}] 
assert (SYS ; Deadlock) :[deadlock free [FD] ]

Listing 23 shows the CSP code that is equivalent to the UML diagram seen earlier. By convention channel declarations come first, followed by the constants that are defined to hold the parameters of this stunt instance. The functions come next, with their accompanying preprocessor keywords. The CSP becomes interesting in the decision processes. Each decision is carried out independently by evaluating the results of functions, and conditionally producing a boolean result. The result of each decision is output onto a channel, which is interpreted by the Decision process. This process runs in parallel with Combined, which by internal choice allows the decisions to occur in any order. Only once both decisions have occurred will the Overall process run, which begins a process that outputs whether or not the stunt can be allowed to go ahead.

The correctness of this decision is clearly a safety critical one, so we have carried out some verification to ensure that not only an answer is always produced, but that it is deterministic. The tool FDR2 can automatically check for live lock and deadlock, which is carried out in the first two assert statements. To check for liveness we need an additional process, which we define with internal choice to express that either the goAhead or endStunt action will occur. This is an example of an Oracle, because a decision is reached without the process requiring any knowledge of how it is achieved. We check to see if the Decision process is a failures refinement of the Liveness process, in doing so hiding the actions in which each decision is undertaken. The deadlock test similarly required an additional process, in this case to prevent FDR2 from showing deadlock on the skip termination event. The Deadlock process is defined as one that infinitely carries out an action, which we have sequentially composed with the SYS process. We know that it will not run until SYS has terminated so deadlock verification is still valid. Figure 23 shows the output from FDR2 after checking all assert statements.

Before synthesis is carried out we must make one change to the script to take account of a limitation in the CSP++ translator. The original designers made a decision not to support the internal choice (|\~|) operator, as modelling non-determinism in an executable is not usually required. For this reason we must remove the Liveness process and associated assert statement. However, once verification is complete this process has no role in evaluating the output, so there is no loss of functionality associated with removing it.

Compiling and executing the translated code will result in the output of either goAhead or endStunt, depending on the values assigned to each of the constants. There is an obvious limitation to the implementation we have presented here: updating the script for a new instance of the stunt requires us to carry out translation and compilation again. To avoid this we would have to create some way of inputting the data from the terminal into the system. This is easily within the scope of the project and could be carried out with user coded functions, just as it
Figure 23: The CSP script modelling the concrete coffin undergoing verification.

was in the previous sorting example. However, having unbounded integer values coming in to the system makes verification very difficult as the state space increases, and for this reason we have not implemented it here. The translated source code is given in Appendix G.

4 Evaluation

This section is a critical discussion of how the events of the project progressed. We consider three aspects of the project: the contribution to the field, the logistical management of the project, and opportunities remaining for development beyond what this project has achieved.

4.1 Scientific contribution

The CSP++ system and this project sits firmly within the formal systems software engineering field. In the inception of the tool, Professor Gardner envisioned CSP++ to be a way of bringing formal methods into wider use, particularly in industry where their acceptance is very low, usually only appearing where their application is mandated. By applying selective formalism and translation it was hoped that more verified software products would be created.

We cannot offer any insight into whether or not this vision has become reality, but we have confidence the work carried out in this project has brought it closer to being reality. The developments and additions we have made to the translator and to the object oriented application framework library have allowed a much wider subsection of allowed CSP specifications to be translated. This means that it becomes a more attractive proposition to practitioners who perhaps have existing CSP specifications that were previously outside the remit of our translator - as were some of the examples we translated above.

Out of our project aims we achieved all but one, which was the addition of tuples to the translatable constructs. The decision was made to omit these because of difficulties encountered when we began work on user defined functions. In discussion with Professor Gardner it was decided that the translation of functions would be an asset of far greater value, so the additional time was spent here instead.
4.2 Project management

Although on the whole our activity has lead to the successful and timely delivery of our product, the process leading to this stage has not been uneventful. During the early parts of the project our aims were set out, and a clear path both in time management and technical steps were presented in order to achieve them.

During this project the author spent two months working at the Fraunhofer FIRST institute in Adlershof, Berlin, under the supervision of Professor Dr Holger Schlingloff. The disruption caused be relocating to Germany did not affect the project outcome, and the additional perspective and experience offered by the supervision there was an asset whilst the parsing process was understood.

The most significant challenge was experienced when our initial industrial sponsor elected to withdraw from supporting the project midway through its course, for reasons independent of advancement and outcome of the work. Without the support of an appropriate industrial partner the project was no longer eligible for the funding received from the European Social Fund, so we were fortunate to receive short notice yet whole hearted support from Professor Jon Holt, a representative of the critical systems company Atego.

Atego have been very satisfied with the outcome of the project, and have added their own words to this section, which we have added in Appendix H.

4.3 Future work

Although much has been achieved both before and during this project, this is not the end of the line for CSP++. An obvious place to begin a further project would be with our unsuccessful aim, to add tuples to the subset of CSP\textsubscript{M} which can be translated. This is certainly possible, and would be a straight forward development of the work presented here.

Beyond this, the specifications that CSP++ can translate are, as just mentioned, a subset of the CSP\textsubscript{M} grammar. Any steps taken to expand this to ultimately support all constructs available to the authors of CSP models would be positive, and make the tool more usable in everyday applications. This would, of course, move the project ever closer to achieving the initial vision.

In the introduction we identified a development paradigm that can be applied, running from a model in UML to an executable program linked with some user coded functions. It would be an interesting step to work with an automatic UML to CSP translator, feeding output from the first step directly into verification and on into the CSP++ translator. Expressing verification goals in the UML model would allow this to progress fully automatically, and could be another step toward automating the development of safety critical systems.

5 Conclusion

Although not without its challenges, we are satisfied that this project has been a success. From a scientific point of view the range of constructs that can be safely translated from CSP to C++ has been expanded, and in turn this has expanded the range of scripts that can be translated successfully and correctly to executable output. With respect to the aims we stated at the beginning of the project, it can be seen from our examples that we have achieved what we set out to do.

From an industrial perspective, preconceptions of what is possible in a development life cycle have been challenged, which may in turn lead to new ideas. In our escapist example we showed that formal development is possible from a UML requirements specification, through to a tool that could be easily used in the field.
From a project management perspective the project was also successful. Although the key
difficulty was encountered when the initial industrial sponsor withdrew support from the project
this was swiftly overcome, and did not affect the schedule of work. The exclusion of tuples from
the final version of the translator is regrettable, but the decision was not made lightly, and we
feel does not adversely affect the outcome of the project in any grave way.

We feel this area has much potential for the future, and are confident that CSP++ can
continue to grow into a mature tool, ready for use in academic and workplace scenarios.
References


A Modified CSP\textsubscript{M} code for CSP++ input

```csp
channel serverReceiveAlert, getDriverPhoneData, callDriver, assessDescription
channel cancelAlert, getMapLocation, processAlert, getServiceFormat
channel processJoin, createServiceDescription

S1 = serverReceiveAlert -> S2
S2 = getDriverPhoneData -> S3
S3 = callDriver -> S4
S4 = if (false) then M1 else ( if (true) then D2 else D1a )
M1 = C1
D2 = DM
D1a = assessDescription -> D1b
D1b = DM
DM = if (true) then D3 else M2
M2 = C1
C1 = cancelAlert -> C2
C2 = SKIP
```
D3 = F3 ||| F2 ||| F1
F1 = getMapLocation -> J1
F2 = processAlert -> J2
F3 = getServiceFormat -> J3
J1 = processJoin -> E1
J2 = processJoin -> SKIP
J3 = processJoin -> SKIP
E1 = createServiceDescription -> E2
E2 = SKIP
SYS = S1

B C++ code representing the translated ATM

/∗
Translated by cspt 5.2 @ Tue Apr 24 10:13:17 2012
(CSPm) CSPTranslate.csp >>> (CSP++) CSPTranslate.cc
*/
#include "Lit.h"
#include "Agent.h"
#include "Action.h"
#include "main.h"
int timeunit = 1000; // milliseconds (1 sec)
AGENTDEF( SYS, "SYS", 0 );

#ifdef dispenseCash_p
extern ActionProc dispenseCash_p;
#else
#define dispenseCash_p 0
#endif
static Atomic dispenseCash("dispenseCash", 0, dispenseCash_p);

#ifdef enterPin_p
extern ActionProc enterPin_p;
#else
#define enterPin_p 0
#endif
static Channel enterPin("enterPin", 0, enterPin_p);

#ifdef pinOK_p
extern ActionProc pinOK_p;
#else
#define pinOK_p 0
#endif
static Atomic pinOK("pinOK", 0, pinOK_p);

#ifdef rejectPin_p
extern ActionProc rejectPin_p;
#else
#define rejectPin_p 0
#endif
static Atomic rejectPin("rejectPin", 0, rejectPin_p);

#ifdef setPin_p
extern ActionProc setPin_p;
#else
#define setPin_p 0
#endif
#endif
static Channel setPin( "setPin" , 0 , setPin.p);

AGENTPROC( SYS_ )
FreeVar x;
FreeVar y;

setPin() >> y;
enterPin() >> x;
if (x==y) {
ipOK();
dispenseCash();
CHAIN0( SYS_ );
}
else {
rejectPin();
CHAIN0( SYS_ );
}

int main( int argc , char* argv[] )
{
#ifdef START
#define START SYS_
#endif
MAIN( argc , argv , START );
}

C C++ output from CSP++ example translation

/*
 Translated by cspt 5.1a @ Tue Jul 31 14:33:36 2012

 (CSPm) PostProcessed–UMLtoCSP–CSP.esp >>> (CSP++) UMLtoCSP–CSP.cc
 */

#include "Lit.h"
#include "Agent.h"
#include "Action.h"
#include "main.h"
int timeunit = 1000; // milliseconds (1 sec)

AGENTDEF( C1_ , "C1" , 0 );
AGENTDEF( C2_ , "C2" , 0 );
AGENTDEF( D1a_ , "D1a" , 0 );
AGENTDEF( D1b_ , "D1b" , 0 );
AGENTDEF( D2_ , "D2" , 0 );
AGENTDEF( D3_ , "D3" , 0 );
AGENTDEF( D4_ , "D4" , 0 );
AGENTDEF( E1_ , "E1" , 0 );
AGENTDEF( E2_ , "E2" , 0 );
AGENTDEF( F1_ , "F1" , 0 );
AGENTDEF( F2_ , "F2" , 0 );
AGENTDEF( F3_ , "F3" , 0 );
AGENTDEF( J1_ , "J1" , 0 );
AGENTDEF( J2_ , "J2" , 0 );
AGENTDEF( J3_ , "J3" , 0 );

59
AGENTDEF( M1, "M1", 0 );
AGENTDEF( M2, "M2", 0 );
AGENTDEF( S1, "S1", 0 );
AGENTDEF( S2, "S2", 0 );
AGENTDEF( S3, "S3", 0 );
AGENTDEF( S4, "S4", 0 );
AGENTDEF( SYS, "SYS", 0 );

#ifdef assessDescription
    extern ActionProc assessDescription;
#else
    #define assessDescription 0
#endif
static Atomic assessDescription("assessDescription", 0, assessDescription);

#ifdef callDriver
    extern ActionProc callDriver;
#else
    #define callDriver 0
#endif
static Atomic callDriver("callDriver", 0, callDriver);

#ifdef cancelAlert
    extern ActionProc cancelAlert;
#else
    #define cancelAlert 0
#endif
static Atomic cancelAlert("cancelAlert", 0, cancelAlert);

#ifdef createServiceDescription
    extern ActionProc createServiceDescription;
#else
    #define createServiceDescription 0
#endif
static Atomic createServiceDescription("createServiceDescription", 0, createServiceDescription);

#ifdef getDriverPhoneData
    extern ActionProc getDriverPhoneData;
#else
    #define getDriverPhoneData 0
#endif
static Atomic getDriverPhoneData("getDriverPhoneData", 0, getDriverPhoneData);

#ifdef getMapLocation
    extern ActionProc getMapLocation;
#else
    #define getMapLocation 0
#endif
static Atomic getMapLocation("getMapLocation", 0, getMapLocation);

#ifdef getServiceFormat
    extern ActionProc getServiceFormat;
#else
    #define getServiceFormat 0
#endif
static Atomic getServiceFormat("getServiceFormat", 0, getServiceFormat);

#ifdef processAlert
    extern ActionProc processAlert;
#else
    #define processAlert 0
#endif
static Atomic processAlert("processAlert", 0, processAlert);
extern ActionProc processAlert_p;
#else
#define processAlert_p 0
#endif
static Atomic processAlert("processAlert", 0, processAlert_p);

#ifdef processJoin_p
extern ActionProc processJoin_p;
#else
#define processJoin_p 0
#endif
static Atomic processJoin("processJoin", 0, processJoin_p);

#ifdef serverReceiveAlert_p
extern ActionProc serverReceiveAlert_p;
#else
#define serverReceiveAlert_p 0
#endif
static Atomic serverReceiveAlert("serverReceiveAlert", 0, serverReceiveAlert_p);

AGENTPROC( S1 )

    serverReceiveAlert();
    CHAIN0( S2 );
}

AGENTPROC( S2 )

    getDriverPhoneData();
    CHAIN0( S3 );
}

AGENTPROC( S3 )

    callDriver();
    CHAIN0( S4 );
}

AGENTPROC( S4 )

    if ( false ) {
        CHAIN0( M1 );
    } else {
        if ( true ) {
            CHAIN0( D2 );
        } else {
            CHAIN0( D1a );
        }
    }

AGENTPROC( M1 )

    CHAIN0( C1 );
}

AGENTPROC( D2 )

61
CHAIN0( DM );

AGENTPROC( D1a )
    assessDescription();
    CHAIN0( D1b );

AGENTPROC( D1b )
    CHAIN0( DM );

AGENTPROC( DM )
    if (true)
        
AGENTPROC( D3 )
    
else
    
AGENTPROC( M2 )
    
AGENTPROC( C1 )
    cancelAlert();
    CHAIN0( C2 );

AGENTPROC( C2 )

END_AGENT;

AGENTPROC( D3 )

Agent::compose(3);
Agent* a1 = START0( F3, 0 );
Agent* a2 = START0( F2, 1 );
Agent* a3 = START0( F1, 2 );
WAIT( a1 );
WAIT( a2 );
WAIT( a3 );
END_AGENT;

AGENTPROC( F1 )
    getMapLocation();
    CHAIN0( J1 );

AGENTPROC( F2 )
processAlert();
CHAIN0( J2_ );
}
AGENTPROC( F3_ )

getServiceFormat();
CHAIN0( J3_ );
}
AGENTPROC( J1_ )

processJoin();
CHAIN0( E1_ );
}
AGENTPROC( J2_ )

processJoin();
END_AGENT;
}
AGENTPROC( J3_ )

processJoin();
END_AGENT;
}
AGENTPROC( E1_ )

createServiceDescription();
CHAIN0( E2_ );
}
AGENTPROC( E2_ )

END_AGENT;
}
AGENTPROC( SYS_ )

CHAIN0( S1_ );
}

int main( int argc , char* argv[] )
{
#ifdef START
#define START SYS_
#endif
    MAIN( argc , argv , START );
}

D CSP test scripts

--- TEST: 041
--- SINCE: V5.1
---TARGET: Set integer operators
PASS: Checks set functionality (prints "correct" if all pass)

channel correct, incorrect

SetA = {}
SetB = {1}
SetC = {2, 3}
SetD = {1, 2, 3}
SetE = {2, 3}
SetF = {5, 4, 67, 9, 8568}
SetG = {[5], {4}, {67, 9, 8568}}
SetH = {[2, 4], {2}}
SetI = {1..3}
SetJ = {{1, 2}, {2, 3}, {2}}
SetK = {2}
SetL = {{1, 2}, {2, 3}, {1}}

SYS = if (empty(SetA)) then First else F
First = if (empty(SetB)) then F else Second
Second = if (union(SetA, SetB) == SetB) then Third else F
Third = if (union(SetB, SetC) == SetD) then Fourth else F
Fourth = if (member(2, SetC)) then Fifth else F
Fifth = if (member(5, SetC)) then F else Sixth
Sixth = if (inter(SetC, SetD) == SetE) then Seventh else F
Seventh = if (card(SetD) == 3) then Eighth else F
Eighth = if (card(SetA) == 0) then Nineth else F
Nineth = if (diff(SetD, SetE) == SetB) then Tenth else F
Tenth = if (Union(SetG) == SetF) then Eleventh else F
Eleventh = if (SetD == SetI) then Twelth else F
Twelth = if (Inter(SetJ) == SetK) then Thirteenth else F
Thirteenth = if (Inter(SetL) == SetK) then F else T

T = correct -> SKIP
F = incorrect -> SKIP

--- TEST: 042
--- SINCE: V5.1
--- TARGET: List integer operators
--- PASS: Checks list functionality (prints "correct" if all pass)

channel correct, incorrect

ListA = <>
ListB = <1>
ListC = <2, 3>
ListD = <1, 2, 3>
ListE = <2, 3>
ListF = <1, 2>, <2, 3>
ListG = <1, 2, 2, 3>

SYS = if (null(ListA)) then First else F
First = if (null(ListB)) then F else Second
Second = if (tail(ListB) == ListA) then Third else F
Third = if (tail(ListD) == ListC) then Fourth else F
Fourth = if (elem(2, ListC)) then Fifth else F
Fifth = if (elem(5, ListC)) then F else Sixth
Sixth = if (head(ListD) == 1) then Seventh else F
Seventh = if (concat(ListF) == ListG) then T else F

T = correct -> SKIP
F = incorrect -> SKIP

--- TEST: 043
--- SINCE: V5.1
---TARGET: List boolean operators
--- PASS: Checks list functionality (prints "correct" if all pass)
channel correct, incorrect

ListA = <>
ListB = <true>
ListC = <false, true>
ListD = <true, false, true>
ListE = <<true, false>, <false>>
ListF = <false, true, false>

SYS = if (null(ListA)) then First else F
First = if (null(ListB)) then F else Second
Second = if (tail(ListD) == ListC) then Third else F
Third = if (tail(ListD) == ListC) then Fourth else F
Fourth = if (elem(true, ListB)) then Fifth else F
Fifth = if (elem(false, ListB)) then F else Sixth
Sixth = if (head(ListD) == true) then Seventh else F
Seventh = if (length(ListA) == 0) then Eighth else F
Eighth = if (length(ListD) == 3) then Nineth else F
Nineth = if (concat(ListE) == ListF) then T else F

T = correct -> SKIP
F = incorrect -> SKIP

--- TEST: 044
--- SINCE: V5.1
---TARGET: Set boolean operators
--- PASS: Checks set functionality (prints "correct" if all pass)

channel correct, incorrect

SetA = {}
SetB = {false}
SetC = {true, false}
SetD = {false, true, false}
SetE = {false, true}
SetF = {true}
SetG = {true, false}
SetH = {{true, false}, {false, false}}
SetI = {{true, false}, {}}

SYS = if (empty(SetA)) then First else F
First = if (empty(SetB)) then F else Second
Second = if (union(SetB, SetB) == SetB) then Third else F
Third = if (union(SetB, SetC) == SetE) then Fourth else F
Fourth = if (member(false, SetB)) then Fifth else F
Fifth = if (member(true, SetB)) then F else Sixth
Sixth = if (inter(SetC, SetD) == SetE) then Seventh else F
Seventh = if (card(SetD) == 2) then Eighth else F
Eighth = if (card(SetA) == 0) then Nineth else F
Nineth = if (diff(SetC, SetB) == SetF) then Tenth else F
Tenth = if (Union(SetH) == SetG) then T else F
Eleventh = if (Inter(SetH) == SetB) then T else F

T = correct -> SKIP
F = incorrect -> SKIP

**E  C++ code for the Children’s Puzzle**

/*
 * Translated by cspt 5.1a @ Fri Aug 31 14:17:55 2012
 * (CSPm) PostProcessed-ChildsProblem.csp >>> (CSP++) ChildsProblem.cc
 */

#include "Lit.h"
#include "Agent.h"
#include "Action.h"
#include "main.h"
int timeunit = 1000; // milliseconds (1 sec)

AGENTDEF( Child_c0v, "Child", 2 );
AGENTDEF( Child_c1v, "Child", 2 );
AGENTDEF( Child_c2v, "Child", 2 );
AGENTDEF( SYS, "SYS", 0 );
AGENTDEF ( SYS_s1, "SYS", 0 );
AGENTDEF ( Stable, "Stable", 0 );
AGENTDEF ( StableAfter_v, "StableAfter", 1 );

#ifdef c0_p
  extern ActionProc c0_p;
#else
  #define c0_p 0
#endif
static Channel c0 ( "c0", 0, c0_p );
static ActionRef c0_r ( c0 );

#ifdef c1_p
  extern ActionProc c1_p;
#else
  #define c1_p 0
#endif
static Channel c1 ( "c1", 0, c1_p );
static ActionRef c1_r ( c1 );

#ifdef c2_p
  extern ActionProc c2_p;
#else
  #define c2_p 0
#endif
static Channel c2 ( "c2", 0, c2_p );
static ActionRef c2_r ( c2 );

#ifdef d_p
  extern ActionProc d_p;
#else
  #define d_p 0
#endif
static Atomic d ( "d", 2, d_p );

int fill ( int n ) {
  if ( ( n%2)==0 ) { return n; }
  else { return ( n+1); }
}

AGENTPROC ( Child_c0v )
#define x ARG(1)
FreeVar y;
Agent::startDChoice ( 2 );
c1() << ((x/2));
c0() >> y;
switch ( Agent::whichDChoice() ) {
case 0: {
d(1, (x/2));
c0() >>= y;
  CHAIN2 ( Child_c0v, 0, fill ( ((x/2)+y) ) );
  break;
}
default: {
  c1() << ((x/2));
  d(1, (x/2));
  CHAIN2( Child,c0v, 0, fill(((x/2)+y)) );
  break;
}
}
#undef x
}

AGENTPROC( Child,c1v )
#define x ARG(1)
FreeVar y;

Agent::startDChoice( 2 );
c2() << ((x/2));
c1() >> y;
switch ( Agent::whichDChoice() ) {
  case 0: {
    d(2, (x/2));
    c1() >> y;
    CHAIN2( Child,c1v, 1, fill(((x/2)+y)) );
    break;
  }
  default: {
    c2() << ((x/2));
    d(2, (x/2));
    CHAIN2( Child,c1v, 1, fill(((x/2)+y)) );
    break;
  }
}
#undef x
}

AGENTPROC( Child,c2v )
#define x ARG(1)
FreeVar y;

Agent::startDChoice( 2 );
c0() << ((x/2));
c2() >> y;
switch ( Agent::whichDChoice() ) {
  case 0: {
    d(0, (x/2));
    c2() >> y;
    CHAIN2( Child,c2v, 2, fill(((x/2)+y)) );
    break;
  }
  default: {
    c0() << ((x/2));
    d(0, (x/2));
    CHAIN2( Child,c2v, 2, fill(((x/2)+y)) );
    break;
  }
}
#undef x
\#define n ARG(0)
FreeVar x;

if (n>0) {
    Agent::startDChoice( 3 );
    c0() >> x;
    c1() >> x;
    c2() >> x;
    switch ( Agent::whichDChoice() ) {
        case 0: {
            CHAIN1( StableAfter.v , (n-1) );
            break;
        }
        case 1: {
            CHAIN1( StableAfter.v , (n-1) );
            break;
        }
        default: {
            CHAIN1( StableAfter.v , (n-1) );
            break;
        }
    }
    else {
        CHAIN0( Stable_ );
    }
\#undef n
}

AGENTPROC( Stable_ )
Agent::startDChoice( 3 );
    c0() << 2;
    c1() << 2;
    c2() << 2;
    switch ( Agent::whichDChoice() ) {
        case 0: {
            CHAIN0( Stable_ );
            break;
        }
        case 1: {
            CHAIN0( Stable_ );
            break;
        }
        default: {
            CHAIN0( Stable_ );
            break;
        }
    }
AGENTPROC( SYS_s1 )

c1.r.sync();
{
    Agent::compose( 2 );
    Agent* a1 = START2( Child_c0v, 0, 0, 0 );
    Agent* a2 = START2( Child_c1v, 1, 1, 2 );
    WAIT( a1 );
    WAIT( a2 );
}
Agent::popEnv( 1 );
END_AGENT;

AGENTPROC( SYS_ )

c0.r.sync();
c2.r.sync();
{
    Agent::compose( 2 );
    Agent* a3 = START0( SYS_s1, 0 );
    Agent* a4 = START2( Child_c2v, 1, 2, 4 );
    WAIT( a3 );
    WAIT( a4 );
}
Agent::popEnv( 2 );
END_AGENT;

int main( int argc , char* argv[] )
{
    #ifndef START
    #define START SYS_
    #endif
    MAIN( argc , argv , START );
}

F    C++ and UCF code for the linear sorting array

/*@ Translated by cspt 5.1a @ Sun Sep 2 11:15:49 2012 
(CSPm) PostProcessed-Sorting.csp >>> (CSP++) Sorting.cc */

#include "Lit.h"
#include "Agent.h"
#include "Action.h"
#include "main.h"
int timeunit = 1000; // milliseconds (1 sec)
AGENTDEF( P0 , "P0" , 0 );
AGENTDEF( P1 , "P1" , 0 );
AGENTDEF( P2 , "P2" , 0 );
AGENTDEF( P3 , "P3" , 0 );
AGENTDEF( PM0_vv , "PM0" , 2 );
AGENTDEF( PM1_vv , "PM1" , 2 );
AGENTDEF( PM2_vv , "PM2" , 2 );
AGENTDEF( PM3_vv , "PM3" , 2 );

70
AGENTDEF( SYS_, "SYS", 0 );
AGENTDEF( SYS_s1, "SYS", 0 );
AGENTDEF( SYS_s2, "SYS", 0 );

#ifdef c0_p
  extern ActionProc c0_p ;
#else
  #define c0_p 0
#endif
static Channel c0("c0", 0, c0_p);

#ifdef c1_p
  extern ActionProc c1_p ;
#else
  #define c1_p 0
#endif
static Channel c1("c1", 0, c1_p);
  static ActionRef c1_r( c1 );

#ifdef c2_p
  extern ActionProc c2_p ;
#else
  #define c2_p 0
#endif
static Channel c2("c2", 0, c2_p);
  static ActionRef c2_r( c2 );

#ifdef c3_p
  extern ActionProc c3_p ;
#else
  #define c3_p 0
#endif
static Channel c3("c3", 0, c3_p);
  static ActionRef c3_r( c3 );

#ifdef out0_p
  extern ActionProc out0_p ;
#else
  #define out0_p 0
#endif
static Channel out0("out0", 0, out0_p);

#ifdef out1_p
  extern ActionProc out1_p ;
#else
  #define out1_p 0
#endif
static Channel out1("out1", 0, out1_p);

#ifdef out2_p
  extern ActionProc out2_p ;
#else
  #define out2_p 0
#endif
static Channel out2("out2", 0, out2_p);

#ifdef out3_p
  extern ActionProc out3_p ;
#else
#define out3_p 0
#endif

static Channel out3("out3", 0, out3_p);

AGENTPROC( P0 )
FreeVar x;

   c0() >> x;
   CHAIN2( PM0_vv, x, 1);
}

AGENTPROC( PM0_vv )
#define k ARG(1)
#define s ARG(0)
FreeVar x;

   if (k==4) {
      out0() << s;
   }
   else {
      c0() >> x;
      if (x>s) {
         c1() << x;
         CHAIN2( PM0_vv, s, (k+1));
      }
      else {
         c1() << s;
         CHAIN2( PM0_vv, x, (k+1));
      }
   }

END_AGENT;
#undef k
#undef s

AGENTPROC( P1 )
FreeVar x;

   c1() >> x;
   CHAIN2( PM1_vv, x, 1);
}

AGENTPROC( PM1_vv )
#define k ARG(1)
#define s ARG(0)
FreeVar x;

   if (k==3) {
      out1() << s;
   }
   else {
      c1() >> x;
      if (x>s) {
         c2() << x;
         CHAIN2( PM1_vv, s, (k+1));
      }
      else {
         c2() << s;
         CHAIN2( PM1_vv, x, (k+1));
      }
   }
AGENTPROC( P2 )
FreeVar x;

c2() >> x;
CHAIN2( PM2_vv, x, 1);

AGENTPROC( PM2_vv )
#define k ARG(1)
#define s ARG(0)
FreeVar x;

if (k==2) {
    out2() << s;
} else {
    c2() >> x;
    if (x>s) {
        c3() << x;
        CHAIN2( PM2_vv, s, (k+1));
    } else {
        c3() << s;
        CHAIN2( PM2_vv, x, (k+1));
    }
}

AGENTPROC( P3 )
FreeVar x;

c3() >> x;
CHAIN2( PM3_vv, x, 1);

AGENTPROC( PM3_vv )
#define k ARG(1)
#define s ARG(0)

out3() << s;
END_AGENT;

AGENTPROC( SYS_s2 )
c1_r.sync();
{
    Agent::compose( 2 );
Agent* a1 = START0( P0, 0 );
Agent* a2 = START0( P1, 1 );
WAIT( a1 );
WAIT( a2 );
} Agent::popEnv( 1 );
END_AGENT;
}
AGENTPROC( SYS_s1 )
c2.r.sync();
{
    Agent::compose( 2 );
    Agent* a3 = START0( SYS_s2, 0 );
    Agent* a4 = START0( P2, 1 );
    WAIT( a3 );
    WAIT( a4 );
} Agent::popEnv( 1 );
END_AGENT;
}
AGENTPROC( SYS_ )
c3.r.sync();
{
    Agent::compose( 2 );
    Agent* a5 = START0( SYS_s1, 0 );
    Agent* a6 = START0( P3, 1 );
    WAIT( a5 );
    WAIT( a6 );
} Agent::popEnv( 1 );
END_AGENT;
}

int main( int argc, char* argv[] )
{
    #ifndef START
    #define START SYS_
    #endif
    MAIN( argc, argv, START );
}

#include <iostream>
#include <list>
#include "Action.h"
#include "Lit.h"

std::list<int> numbers;
int output[4];
bool begin = true;
int counter = 0;

bool done1, done2, done3, done4 = false;

void readInToArray()
{
    std::cout << std::endl << std::endl
    << "Begin reading into four position list ..." 
    << std::endl;
}
```cpp
int a, b, c, d;

std::cout << "Enter integer 1: " << std::endl;
std::cin >> a;

std::cout << "Enter integer 2: " << std::endl;
std::cin >> b;

std::cout << "Enter integer 3: " << std::endl;
std::cin >> c;

std::cout << "Enter integer 4: " << std::endl;
std::cin >> d;

numbers.push_back(a);
numbers.push_back(b);
numbers.push_back(c);
numbers.push_back(d);

std::cout << std::endl;
std::cout << "The list looks like this: {";
std::list<int>::iterator it;
for(it = numbers.begin(); it != numbers.end(); it++) {
    if(it != numbers.begin()) std::cout << ", ";
    std::cout << *it;
}
std::cout << "}" << std::endl << std::endl;
}

void checkFinished() {
    if (done1 & done2 & done3 & done4) {
        std::cout << "The sorted list looks like this: {";
        std::cout << output[0] << ", " << output[1] << ", ";
        << std::endl << std::endl;
    }
}

void getInput( ActionType aT
    , ActionRef * r
    , Var * v
    , Lit * l ) {

    if (begin) {
        readInToArray();
        begin = false;
    }

    *v = Lit( numbers.front() );
    numbers.pop_front();
}

void output0( ActionType aT
    , ActionRef * r
    , Var * v
    , Lit * l ) {

    output[0] = *l;
    done1 = true;
```
checkFinished();

void output1 ( ActionType aT,
              ActionRef * r,
              Var * v,
              Lit * l ) {

    output[1] = *l;
    done2 = true;
    checkFinished();
}

void output2 ( ActionType aT,
              ActionRef * r,
              Var * v,
              Lit * l ) {

    output[2] = *l;
    done3 = true;
    checkFinished();
}

void output3 ( ActionType aT,
              ActionRef * r,
              Var * v,
              Lit * l ) {

    output[3] = *l;
    done4 = true;
    checkFinished();
}

G   C++ code for the concrete coffin example

/*
 Translated by cspt 5.1a @ Sun Sep 2 10:41:45 2012

(CSPm) PostProcessed-EscapistV2.csp >>> (CSP++) EscapistV2.cc
*/

#include "Lit.h"
#include "Agent.h"
#include "Action.h"
#include "main.h"

int timeunit = 1000; // milliseconds (1 sec)

AGENTDEF( BreathEnd_, "BreathEnd", 0 );
AGENTDEF( BreathGo_, "BreathGo", 0 );
AGENTDEF( Combined_, "Combined", 0 );
AGENTDEF( Deadlock_, "Deadlock", 0 );
AGENTDEF( Decision_, "Decision", 0 );
AGENTDEF( Decision_s1_, "Decision", 0 );
AGENTDEF( Decision1_, "Decision1", 0 );
AGENTDEF( Decision2_, "Decision2", 0 );
AGENTDEF( End_, "End", 0 );
AGENTDEF( EquipmentEnd_, "EquipmentEnd", 0 );
AGENTDEF( EquipmentGo_, "EquipmentGo", 0 );
AGENTDEF( Go_, "Go", 0 );
AGENTDEF( Overall_vv, "Overall", 2 );
AGENTDEF( SYS, "SYS", 0 );

#ifdef action_p
  extern ActionProc action_p;
#else
  #define action_p 0
#endif
static Atomic action("action", 0, action_p);

#ifdef breathDecision_p
  extern ActionProc breathDecision_p;
#else
  #define breathDecision_p 0
#endif
static Channel breathDecision("breathDecision", 0, breathDecision_p);
  static ActionRef breathDecision_r(breathDecision);

#ifdef endStunt_p
  extern ActionProc endStunt_p;
#else
  #define endStunt_p 0
#endif
static Atomic endStunt("endStunt", 0, endStunt_p);

#ifdef equipmentDecision_p
  extern ActionProc equipmentDecision_p;
#else
  #define equipmentDecision_p 0
#endif
static Channel equipmentDecision("equipmentDecision", 0, equipmentDecision_p);
  static ActionRef equipmentDecision_r(equipmentDecision);

#ifdef goAhead_p
  extern ActionProc goAhead_p;
#else
  #define goAhead_p 0
#endif
static Atomic goAhead("goAhead", 0, goAhead_p);

#define CoffinHoleLength 10
#define CoffinHoleWidth 4
#define CoffinHoleHeight 4
#define CoffinHeight 3
#define CoffinWidth 2
#define CoffinLength 7
#define CoffinCrushPressure 200
#define Gravity 10
#define FluidDensity 10
#define EscapePumpRate 20
#define EscapologistBMax 2300

int Volume(int lengthParam, int width, int height) {
    return ((lengthParam*width)*height);
}

int Mass(int density, int gravity) {
    return (density*gravity);
}

int Force(int mass, int acceleration) {
    return (mass*acceleration);
}

int SurfaceArea(int width, int lengthParam) {
    return (width*lengthParam);
}

int Pressure(int force, int area) {
    return (force/area);
}

int FillTime(int volume, int pumpRate) {
    return (volume*pumpRate);
}

AGENTPROC( Decision1 )
    if (EscapologistBMax>FillTime(
        Volume(CoffinHeight, CoffinWidth, CoffinLength),
        EscapePumpRate)) {
        CHAIN0( BreathGo );
    } else {
        CHAIN0( BreathEnd );
    }
}

AGENTPROC( BreathGo )
    breathDecision() << true;
    END_AGENT;
}

AGENTPROC( BreathEnd )
    breathDecision() << false;
    END_AGENT;
}

AGENTPROC( Decision2 )
    if (Pressure(Force(Mass(FluidDensity, Gravity), Gravity),
       Gravity)<CoffinCrushPressure) {
        CHAIN0( EquipmentGo );
    } else {
        CHAIN0( EquipmentEnd );
    }
AGENTPROC( EquipmentGo )
    equipmentDecision() << true;
END_AGENT;
}

AGENTPROC( EquipmentEnd )
    equipmentDecision() << false;
END_AGENT;

AGENTPROC( Go )
    goAhead();
END_AGENT;

AGENTPROC( End )
    endStunt();
END_AGENT;

AGENTPROC( Decision_s1 )
    Agent::compose( 2 );
    Agent* a1 = START0( Decision1, 0 );
    Agent* a2 = START0( Decision2, 1 );
    WAIT( a1 );
    WAIT( a2 );
END_AGENT;

AGENTPROC( Decision )
    breathDecision_r.sync();
    equipmentDecision_r.sync();
{
    Agent::compose( 2 );
    Agent* a3 = START0( Decision_s1, 0 );
    Agent* a4 = START0( Combined, 1 );
    WAIT( a3 );
    WAIT( a4 );
}
    Agent::popEnv( 2 );
END_AGENT;

AGENTPROC( Combined )
FreeVar bd;
FreeVar ed;

    Agent::startDChoice( 2 );
    breathDecision() >> bd;
    equipmentDecision() >> ed;
switch ( Agent::whichDChoice() ) {
    case 0: {
equipmentDecision() >> ed;
CHAIN2( Overall_vv, bd, ed );
break;
}

default: {
breathDecision() >> bd;
CHAIN2( Overall_vv, bd, ed );
break;
}
}

AGENTPROC( Overall_vv )
#define bd ARG(0)
#define ed ARG(1)
    if ( bd&&ed ) {
        CHAIN0( Go_ );
    } else {
        CHAIN0( End_ );
    }
#undef bd
#undef ed

AGENTPROC( SYS_ )
    CHAIN0( Decision_ );
}

AGENTPROC( Deadlock_ )
    action();
    CHAIN0( Deadlock_ );
}

int main( int argc, char* argv[] )
{
    #ifndef START
    #define START SYS_
    #endif
    MAIN( argc, argv, START );
}
4th September 2012

To whom it may concern,

In my role as Global Head of Systems Engineering at Atego Systems, the leading provider of mission-critical systems engineering tools and services, I am responsible for thought leadership and research in several areas of systems engineering. As part of my work, I have established a number of excellent working relationships with some of the top universities in the UK, including the Department of Computer Science at Swansea University. It is through this relationship that Atego was invited to contribute to this project.

Initially, the project was supported by a different company who, for whatever reason, withdrew from their role as industrial sponsor. This was very much a case of ‘their loss was Atego’s gain’ as we were fortunate to get involved in what turned out to be an excellent project. The whole topic of formal methods is of interest to Atego and our current research projects – indeed, we are industrial partners for an EU-funded project that is concerned with the application of formal methods to semi-formal modelling techniques, so this project was to fit in perfectly with our research goals.

I was introduced to Daniel as he was nearing the end of his project as he needed an industrial case study that could be used to validate his research.

Atego provided Daniel with a set of diagrams from one of our existing standard models that is used in the company for training and consultancy purposes. Daniel was able to take this partial model and to generate CSP from it, which could then be used to verify the original diagrams and also to be used as a basis for generating C++ code.

As a result of Daniel’s excellent work, Atego now has a complete example of how we can bridge the gap between industrial best practice, such as SysML diagrams into cutting-edge formal research. Extracts from Daniel’s work will be used to enhance our existing training material and feed directly into one of our EU-funded research projects.

As with many successful projects, I only wish that there was more time available as Daniel really provided a proof-of-concept that has the potential to be expanded upon in future projects. My only regret on the project is that Atego was not involved at the earlier stages of the project as I feel that we could have gained even more benefit had we been involved with specifying the project scope.

Overall, this was an outstanding project and I have no hesitation in lending my full support to this work.

Yours faithfully,

Prof Jon Holt
Global Head of Systems Engineering, Atego Systems