



Technical Report
TR-UG-CIS-2009-001

CSP++: An Open Source Tool for Building
Concurrent Applications from CSP Specifications

W.B. Gardner, J. Moore-Oliva, J. Carter, A. Guntie, Y. Solovyov
Dept. of Computing and Information Science
University of Guelph

Submitter: William Gardner

January 1, 2009

Computing and Information Science
University of Guelph
Guelph, Ontario, N1G 2W1Canada
Phone: 519 824 4120 x52483
Fax: 519 837 0323
Web: <http://www.cis.uoguelph.ca>

© 2009 W.B. Gardner, J. Moore-Oliva, J. Carter, A. Gumtie, Y. Solovyov

Statements:

1. This is a non-edited technical report submitted by faculty, technical staff, and/or undergraduate/graduate student(s) in the Department of Computing and Information Science, University of Guelph. The author or authors are responsible for its contents. For any concerns or questions, please contact the author(s) AND the department Chair (chair@cis.uoguelph.ca).
2. The work has not been published before.
3. This may be a report about a preliminary research result or a pre-published version of the final result publication. A revised version may be published in the future.

Citation:

Technical Report TR-CIS-UG-2009-001, Department of Computing and Information Science, University of Guelph, Canada. http://www.cis.uoguelph.ca/department/technical_reports.html, 2009.

Technical reports published by the Department of Computing and Information science, University of Guelph are freely available via the Internet:
http://www.cis.uoguelph.ca/department/technical_reports.html

CSP++: An Open Source Tool for Building Concurrent Applications from CSP Specifications

W.B. Gardner, J. Moore-Oliva, J. Carter, A. Gumtie, Y. Solovyov

Department of Computing and Information Science, University of Guelph, ON, Canada
{gardnerw,jmooreol,jcarter,agumtie,ysolovyov}@uoguelph.ca

Abstract

The CSP++ object-oriented application framework, which implements CSP execution semantics based on Pth multithreading, is released as open source in C++. The new “micro” version, more suitable for embedded platforms, can be built using GNU autotools, and comes with a suite of regression tests. An Eclipse plug-in is available for application developers. It features a syntax highlighter for CSPm specifications, and is integrated with the FDR2, ProBE, and Checker tools from Formal Systems. The design flow from CSPm input, through formal verification, translation with cspt, integration of user-coded functions, execution with the CSP++ framework, and trace refinement checking will be demonstrated.

1. Introduction

The process algebra CSP (Communicating Sequential Processes) [1], with its formal semantics of interprocess synchronization and communication, is an attractive specification notation for modeling concurrent systems. CSP++ [2] is an object-oriented application framework (OOAF) that provides the means to make specifications written in CSP executable on Unix-variant platforms. The OOAF is now available as open source. This paper describes the new “micro” version (V4.2), with its accompanying Eclipse plug-in, for the benefit of researchers and developers who wish to use it, modify it or adapt it to their diverse purposes.

The original tool chain consists of two components: (1) the runtime library, just described; and (2) the translator that produces a “customization” of the OOAF by outputting C++ source code which instantiates framework classes. The translator, called cspt, now accepts CSPm [3], the same dialect used by the popular commercial verifications tools, FDR2, ProBE, and checker, from Formal Systems Europe Limited. While cspt provides a convenient way to customize an application, it is not

strictly necessary. Code utilizing the OOAF can be produced by hand or by other software synthesis tools.

Compiling the translated C++ source code and linking with the framework’s library results in a program that produces the effect of executing the CSP specification. For non-trivial applications, the CSP can be considered as the “control backbone,” where all the interprocess communication and synchronization take place. Then, through a concept dubbed “selective formalism,” individual CSP events and channels may be linked to user-coded functions (UCFs) written in C++. UCFs may be used for purposes such as computation (since CSP is not intended to be a full-fledged programming language) that is not judged necessary to be formally specified, performing I/O, or contacting utilities and services in the operating system (OS) or packages such as a database management system. UCFs must not engage in interprocess communication or synchronization, since that could break the formal model. The idea of software synthesis from a formal specification, coupled with selective formalism, builds a bridge, in effect, between the abstraction level of formal methods and the detailed implementation of conventional programming. Furthermore, by reducing the portion of functionality that must be formally specified, one is less likely to encounter state space explosion during verification.

This translation-based approach contrasts with other CSP-inspired libraries, such as JCSP [4] and C++CSP [5]. A programmer using those libraries may obtain the benefits of components whose semantics mirror those of CSP—particularly channel communication—but it is up to the programmer to put the components together in a useful way. Users may not start with a formal specification, or if they do, they are responsible to implement it manually from the library. CSP++, on the other hand, takes the specification as a design model and automatically generates an implementation to reflect it.

Alternative software synthesis tools for CSP are few, one being Raju et al [6]. CSP++ gives a more complete implementation of CSPm [7], is extensible via selective formalism, and is being actively developed.

Heretofore, CSP++ has been distributed in binary

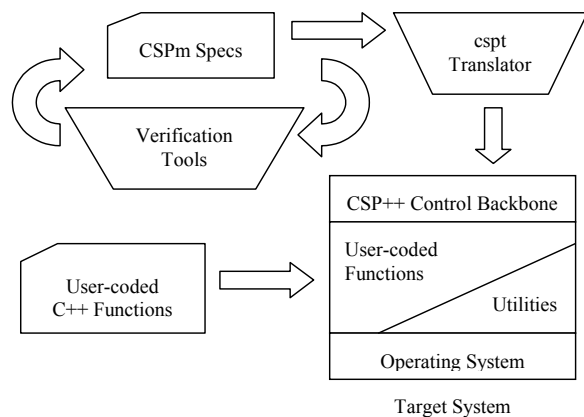


Figure 1. CSP++ Design Flow

form for Solaris (x86) and Linux platforms. It is now available as C++ source code under the GNU Lesser General Public License (LGPL version 2.1). The LGPL, designed for libraries, allows developers to incorporate the existing CSP++ code into their own applications without compelling the resulting “combined work” to become open source. That is, applications based on unmodified CSP++ can be proprietary without violating the license.

Furthermore, developers can modify CSP++ and still produce proprietary combined works, provided they meet this intention and requirement of the LGPL: Modified versions of CSP++, or any such derivative works, must, in turn, be made publicly available as source code under the LGPL. The net result is to expand the benefits to the user community.

The following sections walk through the design flow for application development using CSP++, and provide an overview of the framework’s architecture. Directions for obtaining CSP++ and using the Eclipse plug-in are given. Finally, areas for possible future development are suggested.

2. Design flow

As shown in Figure 1, application development using CSP++ starts with a CSP specification written in CSPm. Verification tools such as those from Formal Systems may be used, and the specification refined, until the developer is satisfied with its suitability and correctness. Next, `cspt` is invoked to translate the CSPm statements to C++ source code. That code is compiled with headers from the OOAF, and linked with the framework’s library. The resulting program will run as a skeleton of the application. The user may interact with it, and direct that a trace—the sequence of all events executed, including actual channel data

passed—be printed. The trace, in turn, can be fed back to FDR2 to check for trace refinement against the CSPm specification.

To integrate UCFs into the application, C++ functions are written and compiled. The translated C++ must be recompiled with preprocessor symbols that specify which events/channels should be associated with which functions. The UCF object files can now be linked with the CSP control backbone and the framework library. When the resulting program is run, the UCFs will be invoked in place of the associated events and channel operations.

The steps of the design flow are greatly facilitated through use of the `cspdt` Eclipse plug-in. It will be described in Section 5.

3. Overview of CSP++ framework

3.1. Object-oriented architecture

In the CSP++ framework, there is essentially a one-for-one mapping of basic CSP elements—process definitions, events, and channels—into runtime objects [8]. Processes are executed by nonpreemptible user-level threads (see below), which are created opportunistically, e.g., for process composition ($P=Q\parallel R$), but not for common tail-recursion cases ($S(i)=a\rightarrow\dots\rightarrow S(i-1)$ or $P=a\rightarrow\dots\rightarrow Q$), and destroyed upon process completion.

In CSP, the meaning of a given event/channel name depends on the context in which its process is invoked, which may include layers and combinations of renaming and/or hiding. To cope with this, the framework maintains a runtime environment “stack” (actually a tree), and each process creation adds a branch to the stack below its parent. When an event/channel reference is encountered during process execution, the stack is searched up through the process’s ancestors to determine the correct identity of the event/channel in context, and to carry out any synchronization required with other processes. Synchronization is complicated by the fact that parties may be involved in choices, whose alternatives need to be rolled back when a choice is resolved.

Events and channels which are not used within the specification for internal synchronization or communication are available for linking with external UCFs. In those cases, the framework will call the UCF, passing it channel output data, or receiving channel input data back into process variables for use within the CSP control backbone.

3.2. Threading support

C++ does not support concurrency in the language, so some add-on solution is required. Since POSIX threads vary from platform to platform as to whether they are preemptible or not—depending on whether the OS maps them directly to kernel-level threads—CSP++ is now based on GNU Pth, Portable Threads. These are true user-level threads within a single OS process, and the Pth installer determines the best way to do context-switching on the target platform. However, Pth does contain provisions for threads to perform I/O that does not block the entire OS process (and with it all other threads).

CSP++ makes limited use of Pth thread features: mutexes, condition variables, and thread-specific data. This situation may change as support for interrupts and timeouts is introduced in a later version.

3.3. Micro version of CSP++

When work was underway to put a CSP++ application—a point-of-sale terminal [9]—on a field-programmable device (FPD), a drawback came to light: the C++ compiler for the FPD did not fully support the Standard Template Library (STL) (which was lightly used in CSP++ for lists and bit vectors) or the `iostream` package (used for printing traces and error messages). Thus the need for a “micro” version of CSP++, more friendly to resource-limited embedded systems, was born.

In V4.2, all uses of STL and `iostreams` have been eliminated. A subset of `std::iostream` functionality is supplied in the new `ucsp` namespace by `istream`, `ostream`, `>>` and `<<` operators, and associated manipulators, all based on `stdio.h`, but with the usual C++ OO interface. Developers are still free to use STL in their user-coded functions, provided their target platform supports it, but CSP++ no longer depends on it.

The micro version also integrated CSP++ with GNU autotools, which makes the source code much easier to build and install onto any combination of hardware platform and Unix-variant operating system.

Furthermore, it includes a new suite of regression tests. Each test is a small CSPm specification. It is translated with `cspt`, compiled, and run. Its output is captured and compared with a “gold” result, ensuring that all of the CSPm operators are being translated and executed properly. The tests ensure that the platform’s combination of the OS, Pth, and the compiler’s runtime routines are performing as expected. Rerunning them is

recommended after making any modifications to the translator or the framework.

4. Obtaining and installing CSP++

CSP++ V4.2 may be downloaded from www.cis.uoguelph.ca/~wgardner, “Research and Downloads” page. The following packages are required (likely present on most systems): flex and bison (or lex and yacc), GNU g++ compiler. These packages need to be obtained: **GNU Pth**, **cppunit**, and **boost**. The latter two are used by regression tests, and CSP++ can be built without them.

Running the supplied `configure` script checks the host environment for package locations and generates custom makefiles. This is the point where build-time options can be injected:

```
./configure CPPFLAGS="-DMEMWATCH  
-DACTWATCH -DERRWATCH"
```

The three preprocessor symbols compile code into the CSP++ installation to enable selective logging on stderr:

- **ERRWATCH**: check for fatal and non-fatal errors at run time, and print messages
- **MEMWATCH**: log all dynamic memory operations concerning framework-managed literals
- **ACTWATCH**: log all searching of the runtime environment stack as event and channel actions are processed

Enabling **ERRWATCH** is a good practice. The other two generate copious output and would normally only be enabled if a problem with CSP++ is suspected, i.e., the application is not generating the trace that the specification requires.

Following the `configure` step, typing “make check” will compile and link all the source files, and run the regression tests.

Typing “make install” will populate three directories in the target location (`/usr/local` by default, can be changed at `configure` step):

- **bin**: `cspt` translator
- **include**: all necessary header files
- **lib**: `libcspxx.a` framework library

Users may also separately download several case studies previously developed by students.

5. Using the Eclipse plug-in

The initial release of an Eclipse plug-in, called **cspdt**, comes with a Users Manual and a Developers

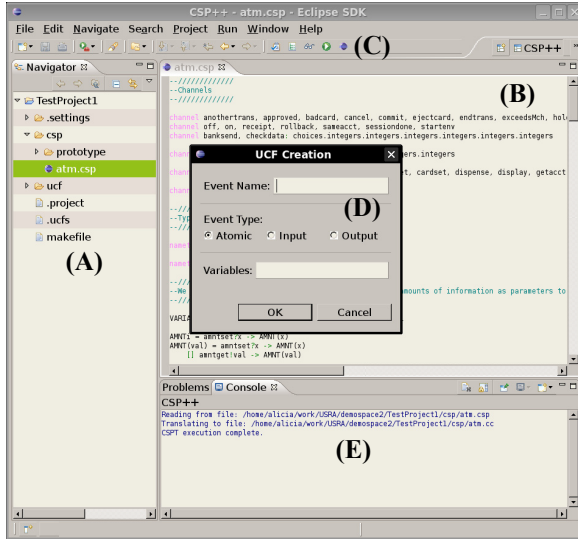


Figure 2. Screenshot of Eclipse plug-in

Manual. The plug-in, compatible with Eclipse 3.2.0, allows a user to create a CSP++ project to be a container for CSPm source code and related files, including makefiles and UCFs. A screenshot is shown in Figure 2. Refer to the lettered callouts in the text below.

Activation of the plug-in is keyed to Eclipse’s recognition of a file type that is open in the editor. The project navigator (A) opens on the left. The source code editor (B) features syntax highlighting for CSPm operators.

When the editor has focus, five buttons (C) appear on the Eclipse toolbar. Assuming that the paths to the Formal Systems tools and to the CSP++ installation have been entered into the Preferences panel, they carry out the following functions with a single click:

- Invoke the Checker tool to validate the CSPm syntax of the open file.
- Start up the graphical ProBE tool, which can be used to explore the state space of the CSPm specification. This is useful for picking up problems like operator precedence mistakes, and checking that the specification means what the user intended.
- Start up the graphical FDR2 tool. It is used to check for deadlocks and to formally verify other assertions that the user may code, such as safety properties.
- Invoke cspt to translate the CSPm specification into a C++ source file, within the project directory.
- Create a user-coded function to link with a specific event or channel. A dialog box (D) will open into which the user enters the event/channel name, its type (input, output, or atomic), and a list of argu-

ments (matching the number of “dotted” components in the channel’s input/output). The plug-in will then create (or append to) a source file `ucfs.cc` in the project containing a C++ skeleton of the UCF. The project’s makefile will also be automatically modified to define the compile-time pre-processor symbol that establishes the runtime linkage between the CSPm event/channel and the corresponding UCF.

For all the above, tool errors are displayed in the Eclipse “Problem” view, linked to and displayed in the editor, and written to the console (E).

Since the program is in C++, the programmer can utilize other features of the Eclipse C++ perspective, e.g., to “make” the project and run the executable. The program can also be debugged with the Eclipse or gdb-based debugger, for example, setting breakpoints and inspecting variables. To make this convenient, cspt intersperses the translated C++ functions with the lines of CSPm from which they arise.

The user may select the command-line option that causes the program to output a trace, and capture it into a text file. An associated tool (Python script) is provided to massage the trace file into a form accepted by FDR2, and then invoke FDR2 to verify the assertion of trace refinement in relation to the CSPm specification.

The use of the csptd plug-in makes the CSP++ design flow much more convenient to follow and less error-prone than carrying out all the steps by typing commands and manually editing files.

6. Future work and conclusion

CSP++ has at least two main uses: Aside from building executable concurrent applications, it also has an educational and training dimension. Students learning CSP could formerly use ProBE to explore their specs and FDR2 to verify properties, but had no easy means to execute them and watch them output a trace. They can do that using CSP++, and establishing trace refinement of the synthesized program with the specification is also a useful experience.

At this point in its evolution, CSP++ is more than an academic toy, but it has areas needing further development before being ready for serious industry use. Suggested targets for future work include the following:

1. Add support for data types beyond integers.
2. Enhance the framework’s interface to UCFs, which is primitive. This is under consideration in conjunc-

tion with adding support for CSPm’s interrupt and timeout operators.

3. Introduce optimization, in the translator and/or via caching, to reduce the stack searching at run time, which penalizes specifications having simple, static process structures.

Numbers 1 and 2, in particular, should be driven by practical needs arising from additional case studies. Meanwhile, support for some operators of Timed CSP [10] will be introduced with the upcoming version 5 of CSP++, which will make the tool more useful for soft real-time systems. The release of CSP++ and the cspt translator under open source licenses may speed up these developments. In the long run, it is hoped that selective formalism will find a place with industry practitioners somewhere on the spectrum between hand-translated formal specifications and a full formal development process that they find unappealing.

Acknowledgment

Creation of the “micro” version of CSP++ and of the Eclipse plug-in was supported by Undergraduate Student Research Awards from Canada’s Natural Sciences and Engineering Research Council.

References

- [1] C. A. R. Hoare. *Communicating Sequential Processes*. Prentice Hall Series in Computer Science. Prentice Hall, Englewood Cliffs, NJ, 1985.
- [2] William B. Gardner. Converging CSP specifications and C++ programming via selective formalism. *ACM Trans. on Embedded Computing Sys.*, 4(2):302–330, 2005.
- [3] S. Doxsee and W.B. Gardner. Synthesis of C++ Software from Verifiable CSPm Specifications. In *12th IEEE International Conference and Workshops on the Engineering of Computer-Based Systems, 2005. ECBS ’05*, pages 193–201, 2005.
- [4] P.H. Welch and J.M.R. Martin. A CSP model for Java multithreading. In *International Symposium on Software Engineering for Parallel and Distributed Systems, 2000. Proceedings.*, pages 114–122, 2000.
- [5] Neil C. Brown and Prof. Peter H. Welch. An Introduction to the Kent C++CSP Library. In Jan F. Broenink and Gerald H. Hilderink, editors, *Communicating Process Architectures 2003*, pages 139–156, 2003.
- [6] V. Raju, L. Rong, and G. S. Stiles. Automatic Conversion of CSP to CTJ, JCSP, and CCSP. In Jan F. Broenink and Gerald H. Hilderink, editors, *Communicating Process Architectures 2003*, pages 63–81, 2003.
- [7] W. B. Gardner. CSP++: How Faithful to CSPm? In *Proc. Communicating Process Architectures 2005 (WoTUG-27)*, Concurrent Systems Engineering, pages 129–146, Eindhoven, September 2005. IOS Press.
- [8] William B. Gardner and Micaela Serra. CSP++: A framework for executable specifications. In M. Fayad, D. Schmidt, and R. Johnson, editors, *Implementing Application Frameworks: Object-Oriented Frameworks at Work*, chapter 9. John Wiley & Sons, 1999.
- [9] John Carter, Ming Xu, and W. B. Gardner. Rapid prototyping of embedded software using selective formalism. In *Proc. 16th IEEE International Workshop on Rapid System Prototyping (RSP’05)*, pages 99–104, Washington, DC, USA, 2005. IEEE Computer Society.
- [10] Steve Schneider. *Concurrent and Real-time Systems: The CSP Approach*. John Wiley & Sons, Ltd., 2000.