Bridging CSP and C++ with Selective Formalism and Executable Specifications

W. B. Gardner
Dept. of Computing & Information Science, Univ. of Guelph, Guelph, Ontario, Canada
wbgardner@alum.mit.edu

Abstract

CSP (Communicating Sequential Processes) is a useful algebraic notation for creating a hierarchical behavioural specification for concurrent systems, due to its formal interprocess synchronization and communication semantics. CSP specifications are amenable to simulation and formal verification by model-checking tools. To overcome the drawback that CSP is neither a full-featured nor popular programming language, an approach called "selective formalism" allows the use of CSP to be limited to specifying the control portion of a system, while the rest of its functionality is supplied in the form of C++ modules. These are activated through association with abstract events in the CSP specification. The target system is constructed using a framework called CSP++, which automatically translates CSP specifications into C++, thereby making CSP directly executable. Thus a bridge is built that allows a formal method to be combined with a popular programming language. It is believed that this methodology can be extended to hardware/software codesign.

1. Introduction

Concurrent systems present special design challenges due to their complex interactions, both with their environment, as in the case of reactive real-time systems, and internally in terms of synchronization and communication among their constituent processes. This is the case whether they are single-host systems, distributed systems, or embedded systems with hardware and software components. Formal methods have been advocated as a way to verify system properties at the design stage [1], but industry practitioners have not been eager to adopt abstruse mathematical notations, uncommon programming languages, or additional costly engineering process steps [20]. Thus concurrent systems often continue to be designed and tested on an ad hoc basis, particularly in North America.

To create a sort of "Third Way" between pure formalism on the one hand, and unfettered informalism on the other, this research proposes an approach dubbed **selective formalism**. Instead of forcing all design to be channelled through the straitjacket of a formalism, formal design can be selectively concentrated on critical parts of the system, i.e., those which will benefit from modeling and verifying the flow of control and communication. Non-critical components written in a popular language can be integrated alongside the formal model without breaking the formalism. Compared to pure formal methods, selective formalism can yield faster development and can facilitate reuse of IP components. Compared to popular programming, selective formalism offers mathematical model-checking and verifiable properties. It may attract practitioners who cannot afford a wholesale adoption of formal methods.

Putting selective formalism into practice requires three ingredients: (1) a suitable formal notation, ideally one with verification tools readily available; (2) a popular programming language; and (3) a framework for integrating these two. This research[1] combines the process algebra CSP (Communicating Sequential Processes) [15][16][22][23] —selected for its formal model of interprocess synchronization and communication—along with C++. Integration is achieved by automatically translating CSP specifications into executable C++ code, and allowing user-coded C++ components to be plugged into what is in effect a CSP control backbone. Work is underway to enhance this framework to support pluggable hardware IP components, which will effectively extend the CSP backbone across the hardware/software interface.

The following sections give an overview of this approach. A small case study based on a simulated disk server, and results of timing experiments, are presented at the end.

2. The CSP++ Approach

This approach relies on two adaptations of the CSP formalism, to wit, making CSP specifications both *executable* and *extensible*. The first adaptation comes through original software tools that automatically synthesize C++ code from a given CSP specification. The resulting software retains the formal properties that may have been verified

[1]This work is supported by research grants from NSERC (Natural Science and Engineering Research Council) of Canada.

by simulation and verification tools such as FDR [11], and can be executed on a platform that hosts C++ with a POSIX-compliant operating system or kernel executive. The run-time environment that emulates CSP process and channel semantics is called CSP++ [13], and is described in Section 2.3.

The second adaptation comes from allowing CSP's abstract named events (including those used as channels) to be associated with arbitrary user-coded C++ functions. They are invoked by CSP++ whenever a process engages in the associated event. These functions can be employed for a variety of purposes, especially for calculations and data processing for which native CSP, which does not pretend to be a full-featured programming language, is ill-suited. The functions can make system calls and interface with the external environment, but they must not engage in interprocess communication or synchronization. That would represent "going behind the back" of the CSP specification, and would potentially violate the system's verified properties.

The above strategy utilizes CSP for modeling the control structure of a system, and then extends the specification by means of the user-coded functions which flesh out its full functionality. As a side benefit, the total system is modularized along the lines of the CSP events and their associated C++ functions, which could in principle constitute reusable IP components. Automatic synthesis of CSP to C++ is key to making this scheme practical, since it eliminates the Achilles heel of many formal methods as distilled in this question: "After the formal specification has been verified, how do we turn it into executable code?" If the answer is, "Translate it by hand," then there is a danger of losing formal properties and introducing errors in the course of labourious manual refinement.

The sections below expand on the design flow, translator, and run-time framework created for CSP++.

2.1. CSP++ System Model and Design Flow

Using the CSP++ approach, a target software system can be constructed in layers, as shown in Figure 1. The CSP++ control layer is actually the CSP system specification, translated into C++, and linked with the CSP++ run-time framework. Triggered by CSP events, it will call on user-coded functions, which in turn can access the hardware directly, utilize operating system facilities via system calls, and optionally access software packages such as a database management system, graphical user interface tool kit, math libraries, etc. Though not shown in Figure 1, when this work is extended to hardware, it is envisioned that some CSP processes can be allocated to VLSI, field-programmable devices, or hardware portions of SOC. The hardware-allocated processes will activate custom circuits

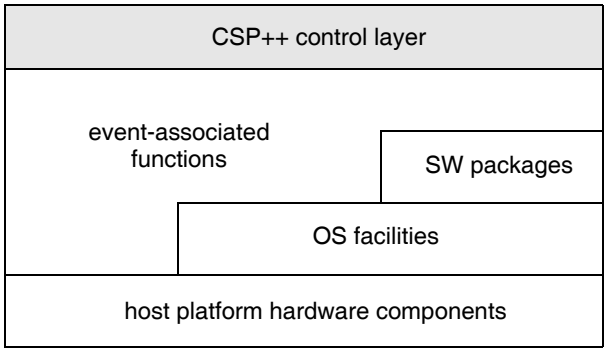

Figure 1. Layered Target System Model

in an analogous manner to which user-coded functions are activated in the software partition. A signalling protocol will allow the semantics of CSP interprocess synchronization and communication to take place across the hardware/software interface.

The design flow for a software system is shown in Figure 2. A designer would start by capturing the desired system behaviour in a CSP specification, relying on verification tools to simulate and check the specification until satisfied. For example, FDR can verify a CSP specification for liveness[2] and for absence of deadlocks. Thereupon, the cspt translator (described in Section 2.2) will convert the CSP statements into C++ source code that is designed to be compiled with the header files of the CSP++ classes (see Section 2.3) and linked with user-coded functions to load onto the target platform. Most likely, the latter functions would form the bulk of the software to be written, and could be delegated to conventional C++ programmers based on the role of these modules in the CSP specification.

Figure 3 clarifies the method of integrating user code. Side (b) is a blowup of the software components on side (a). Some CSP named events (a2, a3, and c1) are used solely for interprocess synchronization and communication within the control layer. Other events (a1, c2, and c3) are associated with user-coded functions. These C++ functions are invoked when a process tries to engage in the associated CSP event.

The translator and the CSP++ framework are described in more detail below.

[2] *Liveness* is the guarantee that "something good will happen," in the sense that a process will engage, as intended, in events that are presented in its environment. In CSP specifications, liveness is verified by analyzing the *refusals* and *failures* of a process, which refer to sets of events offered in the environment that the process can refuse to respond to indefinitely [24].

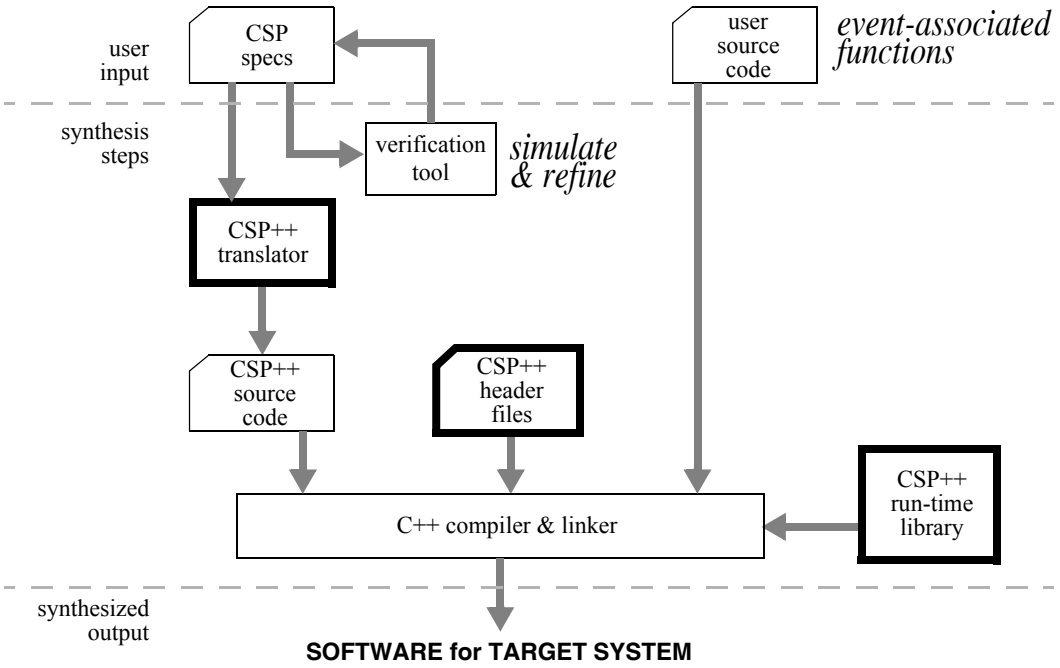

Figure 2. CSP++ Design Flow

2.2. cspt Translator

The CSP++ translator, called cspt, is written in C++ using flex and bison, and takes as input a machine-readable form of Hoare's CSP notation [16]. The input for cspt follows the syntax developed by M. Cheng [6] for use with his model checker called csp12. Its semantics match Hoare's CSP; only the notation has been adjusted to make it convenient for ASCII text file input. Work is underway to realign cspt's input syntax with the commercial model-checker FDR, which uses an alternate machine-readable form of CSP, in order to facilitate bringing "industrial strength" verification tools to bear on realistic case studies.

cspt follows the rules set down in [13] and [12] for translating each CSP construct into one or more C++ statements. These consist of instantiations of classes and opera-

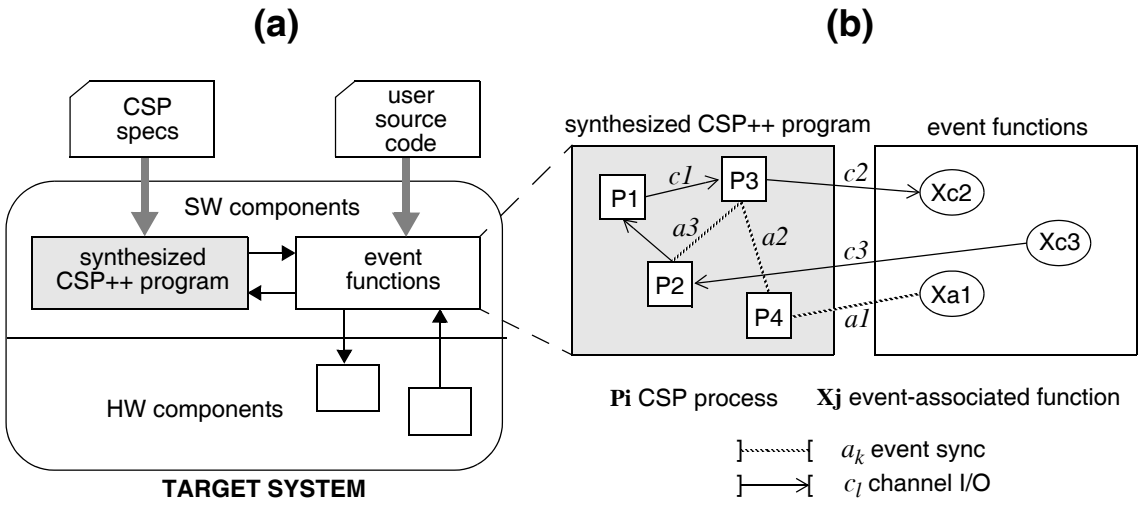

Figure 3. Integration of User Code

tions upon objects of the CSP++ object-oriented application framework (OOAF). The translation of a given CSP specification can be considered a customization of that OOAF yielding a particular application. The customization is an executable C++ program that reflects the translated specification. Running the program (without associated event functions) is equivalent to simulating the specification, and a trace of executed events of synchronizations and channel communication can be logged for printing.

2.3. CSP++ Framework

CSP++ being an object-oriented application framework [18][9][25] means it represents a packaging of OO technology in a way that can be customized by users to create a family of similar applications, in this case, C++ programs which behave like their original CSP specifications. The job of cspt is to automatically generate a customization to match a CSP specification. Like most OOAFs, CSP++ incorporates in its classes some infrastructure needed by all customizations, that is, the CSP computational model consisting of concurrent processes—implemented as threads—"barrier" style synchronization [17], and unidirectional non-buffered channel communication. Classes have been defined for CSP processes (called "agents" in Cheng's terminology), events ("actions"), and channels. The methods of these classes implement their CSP-style behaviours.

The OO design and operation of the framework is described in [13] and fully documented in [12]. Problems solved include implementing concurrent multiparty synchronization in the presence of the CSP external choice operator (|), and proper binding of event names in the dynamic execution environment. CSP allows hiding and renaming of events throughout multiple differing instances of parallel process composition, and this is fully implemented in CSP++.

Briefly, multiparty synchronization is implemented by associating each event with an array of flags, one for each party. When a process attempts to synchronize on an event, it sets its respective flag. If it finds that all flags are now set, then synchronization has occurred; it wakes up the other parties and they go their separate ways. But if any flags remain unset, it sleeps on a condition variable, awaiting wake up by the last party to synchronize. Upon synchronization, if the event is a channel, then data is transferred from the sending process to the receiver.

Binding of event names for the purpose of synchronization and channel communication is performed with the help of a branching environment stack, also called a process descent tree. This tree "pushes" leaf nodes whenever new processes are created (e.g., by the || parallel composition operator), and pops them off when a process terminates. Whenever an event name is referenced in a process, the tree is searched from the process's node in the direction of its ancestors until a definition for the event is found. The definition may specify renaming of the event (in which case the search continues up the tree looking for the new name), hiding (which terminates the search), or synchronization. The flags described in the previous paragraph are physically located in these synchronization nodes of the process descent tree.

External choice, also known as deterministic choice, is implemented by adding a try-then-back-out protocol to the synchronization mechanism. If a process is specified in terms of, say, three alternatives, `a->P1 | b->P2 | c->P3`, then each event, a, b, and c, is tried in turn. The first one to synchronize will determine which process, P1, P2, or P3, is executed. If none are prepared to synchronize, then the choosing process sleeps until the first synchronization occurs. Upon waking up, the pending synchronizations on the other events are cancelled, and the process carries on with the chosen alternative.

When a CSP specification is translated, cspt expects the user to define a top-level process called SYS representing the system as a whole. As a CSP specification is hierarchical in nature, the SYS process, corresponding to the base of the branching environment stack, likely flows via parallel composition into concurrent processes that do the work of the system. Execution ceases when all processes have terminated normally (by executing SKIP), or one has invoked the special STOP (deadlock) process, which by default results in a process status dump.

C++ not being inherently a concurrent programming language, multithreading is obtained via the POSIX thread package. In an earlier version of CSP++, the AT&T task (coroutines) library were utilized. Experience with portability of CSP++ is taken up in Section 3.3, and run-time efficiency in Section 3.2.

3. Discussion

In this section, there is a review of related work in executable specifications, followed by experimental results using CSP++ for a modest case study. Portability of the framework is also discussed.

3.1. Related Work

There are diverse approaches to incorporating elements of formalism into a system design flow. One is to utilize programming languages whose semantics are derived from formal notations. The programming language `occam` has been derived from CSP. [15] shows how to convert from CSP to `occam`, but acknowledges that it is a "very specialized language intended for implementation on transputers."

There is another well-developed derivative of CSP and CCS, Calculus of Communicating Systems [21], called LOTOS [19]. It is similar to `occam` in being a full-featured programming language. In addition to the process-algebraic aspect, LOTOS also incorporates a data-algebraic subset based on abstract data types, and it compiles to executable code. The language has been standardized (ISO 8807), and is in use, particularly in Europe, for design of distributed systems and protocols. In conjunction with using LOTOS as a specification language for hardware/software codesign [5], synthesis tools for translation of LOTOS to C and VHDL have been created. As with `occam`, LOTOS represents a different direction than this work—that of utilizing a special programming language, albeit one based on a design formalism. From a management standpoint, it can be more practical to hire or train a small number of formal practitioners to write the control structure of a system, while the rest of the programmers code in C++ as before, than to contemplate retraining everyone to use an unusual language.

For Java programmers, the JavaPP (Java Plug & Play) Project has created a set of classes called CJT, Communicating Java Threads [14], which are designed to bring CSP-style synchronization and communication to Java programs. Again, this represents a different goal, but does open up an avenue for converting CSP to Java. It should be noted that programs written in `occam`, LOTOS, and Java (with CJT) are not directly verifiable. One would have to write first in CSP, verify the specifications, and then hand translate to the target language, which risks losing the verified properties.

Another approach is used by the EVES tool and the Verdi language (ORA Canada) [8]. Verdi is an executable specification: it is verifiable by EVES and is compilable, but it requires adopting an uncommon language.

Code generation from CSP has been done to some extent using the C language. The CCSP tool [2] provides a limited facility for translating a subset of CSP into C, but it does not directly support the key parallel composition operator (||). Instead, each CSP process becomes a heavyweight UNIX process, and channels are implemented as UNIX sockets. In contrast, the CSP++ approach supports the full functionality of concurrent composition, and is implemented using threads, thus making it practical for a larger range of applications.

In terms of an input formalism for this research, the goals require that the formalism be checkable and synthesizeable. A few algebraic specification languages meet these criteria, including CCS and ACP, Algebra of Communicating Processes [4]. The existence of the sophisticated commercial tool, FDR, means that a path for applying this research in industry can be followed up.

3.2. Experimental Results

[13] introduced a small disk server case study, whose CSP specification (in csp12 syntax) is given in Figure 4. Lines with "%" denote comments. This is the actual source input to the cspt translator. The case study has been run under the original SunOS implementation of CSP++, and with the current Linux and Solaris ports.

The disk server DSS is defined as a composition of two pairs of processes: the scheduler and request queue, which are synchronized on queue events, and the controller and (simulated) disk, synchronized on controller signals (channels dio and dint). The scheduler, DSched, communicates with the controller via channels dci and dco. DSched is specified as a state machine, cycling through the states idle, busy, and check, responding to requests on channel ds and sending acknowledgements. The request queue, DQueue, uses a subscripted process, DQ(_i), to keep track of the number of requests queued, and a two-cell buffer. The BUFF process demonstrates event renaming and hiding.

In order to put CSP++ into perspective as a tool for code generation, and get a reasonable idea of its efficiency, the same disk server model was created using a commercial Statechart synthesis product with a comparable objective to CSP++: ObjecTime Developer, a design automation package for embedded systems, now part of Rational Rose RealTime. Using the disk server case study as a baseline, repetitions were introduced to inflate its execution time to a significantly measurable level. These test cases are laid out in Table 1, along with the average execution time obtained on a 400 MHz Pentium II with 128Mb of memory, running Red Hat Linux 6.2. The g++ compiler used was ecgs-2.91.66, with -O2 optimization. Each test was run five times, and the timings averaged.

`C(1)` and `C(2)` are the two simulated disk client processes. `SYS` is the top-level system specification that CSP++ begins to run. The first case introduces a `Test` process to drive the repetitions. Since Linux deals with POSIX threads directly in the kernel, there is a substantial system time component in the total execution time: 41%. In contrast, when this same test was run on a Solaris system, where the POSIX threads are mostly managed in user space, system time dropped to under 1%.

It was observed that in test (1), 20,000 processes are being created and destroyed as the `Test` process loops composing the clients in `C(1)||C(2)`. In test (2), the 10,000-cycle loop is moved from the Test process down into the clients themselves. A new `syncC` action is introduced in order to synchronize the disk requests in pairs, to avoid overflowing the primitive two-cell disk request queue. Now, `C(1)` and `C(2)` are created only once each. One would expect the thread management overhead to decrease accordingly, and indeed the results show a dra-

```
% DQueue:  disk request queue (with 2-cell buffer)
%   enq!<item>enqueue item
%   deq       dequeue item, followed by:
%     next?_x  next item returned, or
%     empty     empty queue indication
CELL ::= left?_x -> shift -> right!_x ->CELL.
BUFF ::= (((CELL#{right=comm}) || (CELL#{left=comm}))^{comm})\{comm}.

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

DQ(_i) ::= enq?_x -> ( left!_x -> shift-> DQ(_i+1) )
       | deq -> ( (if _i=0 then empty -> DQ(0))
             + fix X.(right?_y -> ( next!_y -> DQ(_i-1) )
                   | shift -> X) ).

% DCtrl:  disk controller
%   dci!start(_cl, _blk)start operation on block <_blk> for client <_cl>
%   dco?fini(_cl, _blk)operation finished
DCtrl ::= dci?start(_i, _blk)-> dio!_blk-> dint -> dco!fini(_i, _blk) -> DCtrl.

% Disk:  disk drive (simulated)
%   dio!_blk  perform disk i/o on block _blk
%   dint            disk interrupt signalled
Disk ::=  dio?_blk -> dint ->Disk.

% DSched:  disk scheduler
%   ds!req(_cl, _blk)client <_cl> requests operation on block <_blk>
%   ack(_cl)  client's operation finished
DSched ::= DS_idle.
DS_idle ::= ds?req(_cl, _blk) -> dci!start(_cl, _blk) -> DS_busy.
DS_busy ::= dco?fini(_cl, _blk) -> ( ack(_cl) -> DS_check )
        | ds?req(_cl, _blk) -> enq!req(_cl, _blk) -> DS_busy.

DS_check ::= deq -> ( empty -> DS_idle
              | next?req(_cl, _blk) -> dci!start(_cl, _blk) -> DS_busy ).

% DSS:  disk server subsystem
DSS ::= (  (DSched || DQueue)^{enq,deq,next,empty}
     ||  (DCtrl || Disk)^{dio,dint}   )^{dci,dco}.
```

Figure 4. CSP Specification for Disk Server

matic drop in execution time. Interestingly, the proportion of system time vs. total is about the same as before (44%).

The purpose of test (3) was to see whether pairwise synchronization was really required. It was not, though removing the extra `syncC` action has hardly any appreciable effect on the timing. Tests (2) and (3) seem to be the "best case" timing that can be obtained for 10,000 repetitions (20,000 simulated disk accesses) by simple process restructuring.

Test (4) cuts the repetitions of test (1) in half to see whether the time for looping is scaling linearly, as one would hope. The results, almost exactly one-half of (1), shows that this is the case.

The model built using ObjecTime Developer 5.2.1 mirrored the CSP specification. (Since the CSP model was originally created from a StateChart diagram, it was a simple matter to revert to the diagram for use with ObjecTime. However, it was noted that inputting the textual CSP specifications was far faster than drawing the equivalent graphical StateChart using ObjecTime.) The tool was used to

Test Case Description	User Secs.	System Secs.	Total Secs.

(1) 20,000 disk accesses in 20,000 process creations
`C(1) ::= ds!req(1,100)->ack(1)->SKIP.`
`C(2) ::= ds!req(2,150)->ack(2)->SKIP.`
`Test(_i) ::=`
`(if _i>0 then ((C(1)|||C(2)); Test(_i-1))) + STOP.`
`SYS ::= (DSS||Test(10000))^{ds,ack(1),ack(2)}.`
	6.33	4.45	10.78

(2) 20,000 disk accesses, synchronized in pairs, in 2 process creations
`C(1,_n) ::=`
`(if _n>0 then ds!req(1,100)->ack(1)->syncC`
`->C(1,_n-1)) + SKIP.`
`C(2,_n) ::=`
`(if _n>0 then ds!req(2,150)->ack(2)->syncC`
`->C(2,_n-1)) + SKIP.`
`Test(_i) ::= (C(1,_i)||C(2,_i))^{syncC}; STOP.`
`SYS ::= (DSS||Test(10000) )^{ds,ack(1),ack(2)}.`
	1.60	1.25	2.85

(3) 20,000 disk accesses; same as (2) but syncC removed from clients `Test(_i) ::= (C(1,_i)|||C(2,_i)); STOP.`	1.65	1.24	2.89
(4) 10,000 disk accesses; same as (1) with `Test(5000)`	3.20	2.16	5.36

Table 1: Timing test results

generate C++ code to run under control of the ObjecTime real-time executive (Micro Run Time System Release 5.21.C.00). It was compiled using Microsoft Visual C++ 6, and executed in a DOS window under NT4. The hardware platform was identical to that used for the CSP++ time trials under Linux.

In order to set up a test case comparable to those above, the test harness behaviour in the outermost block triggered the clients 10,000 times, thus resulting in 20,000 disk requests, as in the CSP++ version. Timing data was obtained by calling the ANSI C clock() function to return the elapsed CPU time at the start and end of model execution. The difference of start and end times of five runs was averaged to get the result of 3.76 CPU seconds.

A key inference coming from the timing data concerns the overhead inherent in the CSP++ OOAF as it is currently implemented. The helpful breakout of user versus system times in Linux shows clearly that the framework's overhead—consisting of thread creation, thread scheduling, mutex locking and unlocking, and condition variable waiting and signalling—is at least 40%. This is therefore a ripe area to target for optimization.

The purpose of the ObjecTime comparison is to show whether the CSP++ execution times are *reasonable* in light of state-of-the-art code generation tools. As a matter of fact, they compare quite favourably with the ObjecTime results. ObjecTime ran faster than test (1), but when the huge amount of gratuitous process creation was cut out in tests (2) and (3), the CSP++ program finished first. Considering that ObjecTime is an expensive commercial tool, having had years to optimize its real-time executive, the result obtained by the initial version of CSP++ running under generic desktop PC Linux is encouraging.

3.3. Portability

The original implementation of CSP++ was on SunOS with CSP processes mapped to nonpreemptible coroutines. The current version uses Linux with Pthreads (preemptible kernel level threads). CSP-to-C++ translation methodology was stable across the port, and no changes were required to the application code (i.e., the CSP specification of the case study).

The introduction of thread preemption made it necessary to institute a locking discipline for shared data structures. These changes were absorbed within the OOAF classes, and no changes were necessitated to the translation algorithms, which illustrates a significant degree of portability from the OOAF approach.

Though more trouble to program, the preemptive model gives user-coded functions the freedom to block with impunity, say for I/O, without causing scheduling of the entire CSP++ system to freeze. This is a major advantage over the coroutines "many-to-one" scheduling model.

Having achieved success with the Linux port, the new version was recompiled, still using g++, on a Sun worksta-

tion under Solaris, which also supports Pthreads. This worked immediately. In contrast, the coroutines-to-LinuxThreads port revealed the boundary between OS-dependent and OS-independent code, and as hoped, the boundary fell, to an overwhelming extent, at the border between the framework classes and their parent classes in the third-party task library.

4. Future Work

As it stands currently, the initial work on CSP++ has several weaknesses:

1. It is based on the original definition of CSP, which does not include timing and interrupts, thus it is impractical to apply to any system that requires taking notice of timing constraints (e.g., timeouts) in its specifications.

2. The machine-readable form of CSP accepted by the current translator is not compatible with the form utilized by the most capable commercial verification tool, FDR.

3. The framework exhibits considerable inefficiency at run time, due to carrying out dynamic binding on every event named in a CSP specification, regardless of whether this generality is needed or not.

4. The interface between CSP specifications and user-coded C++ modules is still too elementary, and does not allow those modules to easily participate in the important CSP choice operation.

Planned future work to remedy these weaknesses include investigating the correct way to incorporate timing and interrupts into the existing OOAF and CSP translator without breaking the formalism. There are formal models of timed CSP [7], and one `occam` -based tool has implemented timing [3]. Conducting static analysis of CSP specifications in order to carry out compile-time event binding will be explored. Bringing the CSP input syntax into conformity with FDR will open up avenues of collaboration and facilitate the creation of benchmarks and case studies for experimental and evaluation purposes.

Under the heading of hardware/software codesign, the OOAF software synthesis approach will be applied to hardware synthesis, using HDL source code for output and field-programmable gate arrays as an experimental platform. CSP specifications can then be partitioned into software- and hardware-targeted portions, with the cspt translator automatically synthesizing the hardware/software interface. It may be possible to create facilities for exploring architectural alternatives at a system level, based on a CSP specification and a library of existing and simulated software and hardware components.

5. Conclusion

Formal specification is normally an “all or nothing” design technique. By allowing selective use of CSP, a relatively approachable formal method, for the parts of a system where verification is important, we make a formal methodology more attractive and less onerous for practitioners. The CSP++ framework becomes a bridge between a formal method and the popular programming language C++.

CSP++ is available on the Wiley CD-ROM [10] (original SunOS coroutines version of framework, lacking cspt translator). The latest version (LinuxThreads with cspt translator) can be downloaded from the author’s website <http://www.cis.uoguelph.ca/~wgardner>.

6. References

[1] V.S. Alagar and K. Periyasamy. *Specification of Software Systems*. Springer-Verlag, 1998.

[2] B. Arrowsmith and B. McMillin. How to program in CCSP. Technical Report CSC 94-20, Department of Computer Science, University of Missouri-Rolla, August 1994.

[3] A. Balboni, W. Fornaciari, and D. Sciuto. Partitioning and exploration strategies in the TOSCA co-design flow. In *Proc. Fourth International Workshop on Hardware/Software Codesign*, pages 62–69, Pittsburgh, March 1996.

[4] J.A. Bergstra and J.W. Klop. Algebra of Communicating Processes with abstraction. *Theoretical Computer Science*, 37(1):77–121, May 1985.

[5] C. Carreras, J.C. López, M.L. López, C. Delgado-Kloos, N. Martínez, and L. Sánchez. A co-design methodology based on formal specification and high-level estimation. In *Proc. Fourth International Workshop on Hardware/Software Codesign*, pages 28–35, Pittsburgh, March 1996.

[6] Mantis H.M. Cheng. Communicating Sequential Processes: a synopsis. Dept. of Computer Science, Univ. of Victoria, Canada, April 1994.

[7] J. Davies and S. Schneider. A brief history of timed CSP. Technical Report PRG-96, Programming Research Group, Oxford University, April 1992.

[8] EVES web site, ORA Canada. http://www.ora.on.ca/eves.html [as of 2/7/03].

[9] Mohamed E. Fayad and Douglas C. Schmidt. Object-oriented application frameworks. *Communications of the ACM*, 40(10):32–38, October 1997.

[10] M. Fayad, D. Schmidt, and R. Johnson, editors. *Implementing Application Frameworks: Object-Oriented Frameworks at Work*. John Wiley & Sons, 1999.

[11] FDR web site, Formal Systems (Europe) Limited. http://www.fsel.com [as of 2/6/03].

[12] William Gardner. CSP++: An Object-Oriented Application Framework for Software Synthesis from CSP Specifications. Ph.D. dissertation, Dept. of Computer Science, Univ. of Victoria, Canada, 2000. http://www.cis.uoguelph.ca/~wgardner/, Research link.

[13] William B. Gardner and Micaela Serra. *CSP++: A Framework for Executable Specifications*, chapter 9. In Fayad et al. [10], 1999.

[14] G. Hilderink, J. Broenink, W. Vervoort, and A. Bakkers. Communicating Java Threads. In *Proc. of the 20th World Occam and Transputer User Group Technical Meeting*, pages 48–76, Enschede, The Netherlands, 1997.

[15] Michael G. Hinchey and Stephen A. Jarvis. *Concurrent Systems: Formal Development in CSP*. McGraw-Hill Book Company, 1995.

[16] C. A. R. Hoare. *Communicating Sequential Processes*. Prentice Hall, 1985.

[17] Bil Lewis and Daniel J. Berg. *Multithreaded Programming with Pthreads*. Sun Microsystems Press, Prentice Hall, 1998.

[18] Ted Lewis, editor. *Object-Oriented Application Frameworks*. Manning Publications Co., Greenwich, CT, 1995.

[19] L. Logrippo, M. Faci, and M. Haj-Hussein. An introduction to LOTOS: Learning by examples. *Computer Networks and ISDN Systems*, 23:325–342, 1992.

[20] Luqi and Joseph A. Goguen. Formal methods: Promises and problems. *IEEE Software*, 14(1):73–85, January 1997.

[21] R. Milner. *Communication and Concurrency*. Prentice Hall, 1995.

[22] A. W. Roscoe. Model-checking CSP. In A. W. Roscoe, editor, *A Classical Mind: Essays in Honour of C. A. R. Hoare*, Prentice Hall International Series in Computer Science, pages 353–378. Prentice Hall, 1994.

[23] A.W. Roscoe. *The Theory and Practice of Concurrency*. Prentice Hall, 1998.

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

[25] Savitha Srinivasan. Design patterns in object-oriented frameworks. *Computer*, 32(2):24–32, February 1999.