Communicating Process Architectures 2005
Jan Broenink, Herman Roebbers, Johan Sunter, Peter Welch, and David Wood (Eds.)
IOS Press, 2005

CSP++: How Faithful to CSPm?

W. B. GARDNER [1]
Dept. of Computing & Information Science, University of Guelph, Canada

Abstract. CSP++ is a tool that makes specifications written in CSPm executable and extensible. It is the basis for a technique called selective formalism, which allows part of a system to be designed in verifiable CSPm statements, automatically translated into C++, and linked with functions coded in C++. This paper describes in detail the subset of CSPm that can be accurately translated by CSP++, and how the CSP semantics are achieved by the runtime framework. It also explains restrictions that apply to coding in CSPm for software synthesis, and the rationale for those restrictions.

Keywords. CSPm, C++, selective formalism

Introduction

CSP++ is a pair of tools—a translator and an object-oriented application framework (OOAF)—that together make CSP specifications both executable and extensible. The initial development goes back to 1999 [1], and was based on a local dialect of CSP called csp12 [2] that was supported by an in-house verification tool of fairly limited capabilities. In order to bring CSP++ to a wider community, and build a straight-line design flow for software synthesis starting from robust commercial verification tools, CSP++ has been recently redeveloped so as to accept input in CSPm, the same machine-readable dialect used by FDR2 and ProBE from Formal Systems (Europe) Ltd. [3].

There is more to CSP++ besides translation and execution of CSPm specifications. It also includes a strategy for practicing formal methods in software engineering, dubbed *selective formalism* [4]. This strategy provides a logical way to combine formal specifications written in CSPm with source code written in the popular programming language C++, without ruining verified properties. It is an unabashed attempt to breech the resistance of software developers to adopting pure formal methods, by offering a sort of pragmatic compromise.

The purpose of this paper is to expose the specific design choices that went into the selection and implementation of the CSPm subset. It begins with an overview of CSP++, first introducing the approach of selective formalism, and then going through the steps of the design flow carried out by the automated tools.

The heart of the paper helps unfold in four sections the answer to the title's question, How faithful is CSP++ to CSPm? If one wants to synthesize software from a CSPm specification, what can one expect, and what is one limited from doing? Do the execution semantics match the traces of the specification, and what happens when non-formal C++ code is linked in? There is a clear rationale for what has and has not been implemented. Convergence with CSPm is described in terms of the operators and constructs that are synthesizeable (as of version 4.1). *Divergence* from CSPm is also discussed in detail. Note

[1] Assistant Professor, Modeling & Design Automation Group, Dept. of Computing & Information Science, University of Guelph, ON, Canada, N1G 2W1. E-mail: wgardner@cis.uoguelph.ca. CSP++ is available for download from the author's website: http://www.cis.uoguelph.ca/~wgardner, Research link.

that this order of presentation mixes together constructs, philosophy, implementation, and limitations.

The main part of the paper ends with a list of platforms that CSP++ is known to run on, and several case studies from which performance measurements have been gleaned. The paper then concludes with a brief review of related work—other CSP frameworks and translators—and plans for future work. Beyond the obvious issue of what features to add to CSP++, we also muse on what it would take to popularize the practice of selective formalism based on CSP.

1. Overview of CSP++

The overview below assumes that the reader is familiar with Communicating Sequential Processes [5] and needs no special justification for choosing CSP as a modeling tool for concurrent systems. The main contribution of CSP++ is software synthesis based on CSP. The first subsection sets the context for this, by explaining what is meant by "selective formalism" and how it applies to CSP++. The next subsection goes through the steps of the CSP++ design flow. Finally, the synthesis tools—the translator and the OOAF—are described chiefly from a user's standpoint. Their internal operation is not described in detail, in order to avoid duplicating documentation available from other sources [6][7].

1.1 Selective Formalism

The notion of selective formalism is based on the oft-observed fact that resistance to the adoption of formal methods in the software industry runs high. This state of affairs is not without rational basis. For example, three practical drawbacks of formal methods are:

1. A company will not likely have on hand many designers who can utilize a formal notation, and will not be eager to retrain its programmers who are already skilled in conventional programming languages.
2. Even if a specification is produced in a formal notation and subjected to verification, the notation will have to be translated, presumably by hand, into a programming language suitable for implementing on the target platform. Aside from the time-consuming and error-prone nature of this manual step, it may not be clear that properties verified in the formal specification could be retained in translated form.
3. Formal notations are more convenient for expressing abstractions above the level of a detailed implementation. Therefore, specifications written in a formal notation will likely need to be supplemented by conventional program code in any case.

It must be acknowledged that the first point above does not match the profile of companies whose clients have forced them to take a "high road" vis-à-vis formal methods, e.g. for the sake of highly safety-critical products. Such companies have their own solutions, and selective formalism could even represent a backward step for them.

The essential compromise of selective formalism is that many benefits of using a formal notation can be obtained without committing to it for building an entire system. It is proposed that the control backbone of a system be specified using a formalism that is well-suited to expressing interprocess synchronization and communication, i.e. CSP, and that this specification be automatically translated to a conventional language, C++. Provision is made for supplementing the translated formal backbone with additional C++ code in a way that does not invalidate the specification's formal properties. The "selective" aspect refers to the designer's decision to describe more or less of the system in CSP, according to the system's characteristics.

This approach goes far to overcoming the three drawbacks:

1. The company will need only a small number of trained "CSP gurus" who can write CSP and run the verification tools. Much of the coding, integration, and testing can be carried out by programmers skilled in C++.
2. Automatic translation is used to render the verified CSP control backbone into compilable C++. The semantics of the resulting executable code match the CSP specification. Thus, the specification is not destined to become an "orphan" in the development process; it can be modified and retranslated on demand.
3. Programmers need not attempt awkwardly to express every algorithm or calculation in CSP, but can use C++ where formal properties are not an issue. Interfacing the system with its actual environment via I/O can be carried out conveniently in C++.

Selective formalism is therefore based on software synthesis, particularly on the ability to make CSP specifications both executable and extensible. The steps of the design flow based on the automated tools are described in the next subsection. These steps are depicted in Figure 1.

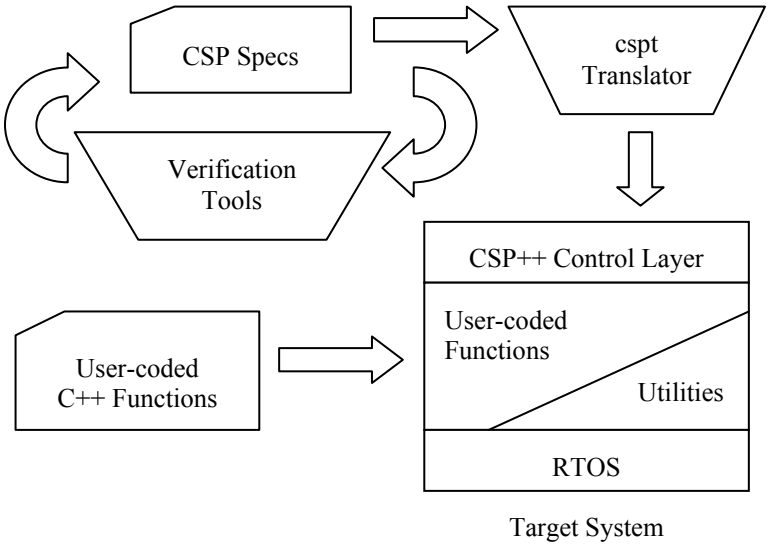

Figure 1. CSP++ Design Flow

1.2 Design Flow

A designer will start by creating a specification in CSPm, and using the tools from Formal Systems—checker, ProBE, and FDR2—to simulate it and verify its properties. Experience shows that CSP tends to feature in four roles in such a specification, constituting four complementary models:

1. *Functional Model:* These statements capture the desired system behavior in terms of CSP processes engaging in named events.
2. *Environment Model:* These statements simulate the behavior of entities in the system's target environment, in terms of processes engaging in events. The functional model can be simulated by synchronizing it with the environment model.
3. *Constraint Model:* Other processes may optionally be added alongside the functional model to limit or constrain the event sequences that can occur. A

constraint model is used to focus on critical event sequences in the functional model that must—or must not—occur in order for the system to be "safe." If verification shows that the constraint is violated, the functional model must be improved.

4. *Implementation Model:* Since the functional model will likely be fairly high-level, it will normally need to be refined to an implementation model, still in CSP, but with more detailed processes and events added. Verification will confirm whether the implementation is a legitimate refinement of the original functional model.

After verification is satisfactory, the CSPm specification can be sent to the synthesis tools (described next), and the resulting C++ source code compiled and linked. This program can be run with tracing enabled for simulation purposes, in which case it will print out a trace of every event executed, identifying the process in control at that moment. (For synchronizing events, only the process that arrived last at the rendezvous will be identified.)

In order to complete the implementation, the designer returns to the CSPm specification and removes (or comments out) the environment model, since the idea is for the translated CSPm to interact with the system's real environment. At this point, named CSPm channels that were previously synchronized with the environment model are now free to be linked with C++ user-coded functions (UCF). These functions can perform system calls, carry out I/O, and utilize third-party packages such as a database management system, under control of the translated CSPm backbone.

For debugging purposes, the translated C++ can be run with a conventional debugger (e.g. gdb). Since the original CSPm is inserted as comments in the translated source file, interleaved with the resulting C++ translation, it is easy to relate the two and, in effect, set breakpoints in the CSPm and inspect local variables. Execution can also be conveniently stepped out of the CSPm into the user-coded functions.

1.3 Synthesis Tools

Since the semantical "distance" between CSP and executable machine code is large, an intermediate code translation target was created in the form of an OOAF. The framework, called CSP++, is architected in terms of C++ classes that mirror the objects in the world of CSP—chiefly processes and channels—and supply their proper semantics. The job of the translator is to convert a CSPm specification into a particular customization of the framework, which, when compiled and run, emulates the original specification. The translator and framework form a tool chain (Fig. 2) and are described in the following subsections.

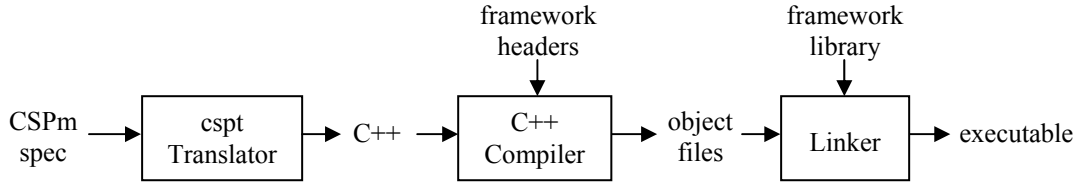

Figure 2. CSP++ Tool Chain

*1.3.1 **cspt** Translator*

The translator, called **cspt**, originally supported the local dialect csp12. For version 4.0, it has been refitted with a front end that accepts a carefully chosen subset of CSPm. The particulars of this subset are discussed in detail in section 2. The aim is that any text file conforming to the subset which is syntactically acceptable to checker, ProBE, and FDR2,

will be translated accurately into C++ source code which, when compiled with the CSP++ class library headers and linked with the CSP++ object library, will execute with the same semantics as simulated by ProBE.

Looking at the output source code, the user will observe that each CSPm process definition has been translated into one or more C++ functions, and that the function bodies contain instantiations and method invocations of CSP++ classes. Some CPP preprocessor macros are used for the translator's convenience, and the readability of the source code is quite high. The process called SYS is taken as the starting point for execution, and a main function is generated to process command line options (e.g. enable tracing) and then launch SYS. Execution stops and the main function returns in any of these circumstances:

1. SYS terminates by executing `SKIP`.
2. Any process executes `STOP`.
3. All processes are waiting for a synchronization event *and* the command line option of idle checking was enabled.

In cases 2 and 3, a dump is printed showing the status of all active processes, identifying which events they are waiting on for synchronization.

1.3.2 Execution Framework

CSPm processes are mapped into threads. The current version of CSP++ is based on GNU Pth [8], a portable package for nonpreemptible threads. The translator is smart enough to avoid consuming resources with gratuitous thread creation: In the two common cases of a process turning into another process (e.g. `P = a -> Q`) and tail recursion (`P = b -> P`), the current thread simply carries on, in the one case changing its identity to Q, and in the recursive case by looping back to P. This is called "chaining."

Compositional cases spawn new threads as required. For example, `P = A||B` would spawn threads for `A` and `B`. The sequence `Q = R;S` would spawn a thread for `R`, wait for it to finish, and then chain to `S` without spawning. Now suppose `R` were written inline, as say, `Q = e->SKIP;S`. In this case no thread would be spawned; e would just be executed by `Q`'s thread.

Complex expressions incorporating composition are handled by extracting unnamed subprocesses. In the example, `Z = (P||Q);R`, `(P||Q)` would be extracted by the translator as a subprocess. `Z` would spawn it to perform the parallel composition and wait for it to finish, after which `Z` would chain to `R`.

Changing the underlying thread model is not difficult, and has been done several times already. The base class for CSPm process objects is called `task`, and all the thread-aware code is localized in its methods for easy portability.

In order to fully emulate the dynamics of a CSPm specification, the runtime system maintains a branching environment stack (i.e. tree structure). Whenever the CSPm elements of synchronization sets, renaming, and hiding are encountered, corresponding environment objects are pushed onto the current process's branch of the stack. All CSPm events are interpreted in light of their process's current environment context, which necessitates a good deal of stack searching.

User-coded functions are integrated as follows: When an event is to be executed, the framework will check whether a user-coded function was supplied at link time. If so, the UCF will be called, and if channel I/O is involved, data will be transferred to/from the UCF. If no UCF is linked to the event/channel name, the event can be used for synchronization with another CSPm process as usual.

The most challenging feature of CSPm to implement is multiparty synchronization in the presence of external choice. This is handled by trying each alternative in turn until one succeeds, or if none succeeds, then suspending the thread on a condition variable. The last party to arrive at a synchronization is called the "active" party. It is responsible for canceling the other choice alternatives (if applicable), transferring any channel data (if applicable), and waking up all remaining "passive" parties.

For simulation purposes, any events that are not synchronized in the specification get some default treatment at run time: plain events and channel output are printed, and integer input is obtained for channel input. This means that, for example, if `P = ch!10`, the framework will output `10`. But if another process is put in parallel with `P`, say, `Q = ch?x`, then nothing will be printed because the event will be absorbed internally.

In addition, as mentioned above, the framework can have trace printing enabled. In that case, each successful synchronization and channel data transfer will be logged on the `cerr` (`stderr`) stream, and will reflect any renaming and/or hiding that is in effect.

2. Convergence with CSPm

Appendix A of the FDR2 User's Manual [9] is taken as the "bible" for CSPm syntax. The same presentation is also available from Appendix B of [9]. The basic principles behind decisions concerning which features of CSPm to support in CSP++ for translation can be stated as follows:

- We want to implement for synthesis a rich, useful subset of CSPm with as few restrictions as possible. Anything one writes in that subset, and verifies, should be synthesizeable without modification and hand-tinkering, since those activities can be fertile sources of bugs.
- The above principle implies that we don't offer "extensions," since those would not be verifiable. Extensions for synthesis' sake that could be camouflaged from FDR2, say as comments, might be entertained in the future.
- We assume that users have access to the Formal Systems tools, so there are some things, such as channel statements, that cspt does not validate. If one bypasses at least running checker before translating, unnecessary problems may be created.

The idea of a "synthesizeable subset" is also found in hardware synthesis. For example, VHDL was originally conceived as a specification language, and then became adapted for simulation. In recent years, CAD vendors have created synthesis products that generate digital circuits from structural or behavioural descriptions input in VHDL. There is no attempt to synthesize each and every VHDL construct, since the language was never created with that intention. Therefore, the vendors define their own synthesizeable subsets of VHDL.

Similarly, CSP++ supports a subset of CSPm for software synthesis. Descriptions of supported constructs are divided below into four areas: events, processes, operators, and other language constructs.

2.1 Supported Events

In CSPm, the events collected into trace sequences are compound symbols made up of components separated by dots. The leftmost symbol is a channel name, and the components to its right (if any) are considered the channel's subscripts and/or data. In CSP++, we dub an event having no data—i.e. a bare occurrence of a channel name—as an *atomic* event.

However, an atomic event may have subscripts. The distinction between subscripts and data in CSPm is blurry; we attempt to clarify it in CSP++ usage (see section 3.3 for full discussion). The designer's intent in using subscripts is likely to define a *group* of channels or events that have the same base name.

CSP++ supports alphanumeric channel names that are accepted by the C++ compiler as valid variable names. Subscripts and data may comprise from 1 to *n* dotted components, where *n* is currently set at 10.

The contents of subscripts and data components are determined by the datatypes supported by the translator. Currently, CSP++ supports only integer data.

2.2 Supported Processes

In CSPm, a powerful feature is the ability to write parameterized process definitions, including multiple definitions of the same-named process. CSP++ supports such overloaded definitions with 0 to *n* parameters, where *n* is currently set at 10. There are two restrictions regarding overloaded process definitions in CSP++:

- All definitions must have the same number of parameters.
- To work as expected, the most general definition should be coded last.

The first restriction means that the set of definitions `P(1)`, `P(2)`, and `P(n)` would be valid in the same specification, but `P`, `P(i)`, and `P(1,n)` would not. The second restriction means that coding `P(n)` before `P(1)` and `P(2)` would result in the `P(n)` definition always being invoked, even by explicit statements such as `a -> P(1)`, which would be contrary to the designer's intent.

The **cspt** translator tells when a process invocation can be resolved at translation time, and when binding must be deferred to run time. In the latter case, a parameter table is generated for any sets of process definitions that require runtime binding.

Process definitions can be recursive, with tail recursion being handled very efficiently. Even infinite tail recursion results in no stack growth.

In terms of special "built-in" process names, `SKIP` and `STOP` are supported. `STOP` aborts execution with a process status dump.

2.3 Supported Operators

CSP++ supports these operators:

- *Prefix:* `event -> proc`
- *Conditional:* `if` *`expr`* `then` *`proc1`* `else` *`proc2`*; where *`expr`* is a relational expression
- *Event renaming*: `proc[[oldname <- newname]]`
- *Event hiding:* `proc\{name}`

CSPm's relational operators (`==`, `!=`, `<`, `>`, `<=`, `>=`) and arithmetic operators (`+`, `-`, `*`, `/`, `%`) are recognized. Renaming and hiding can be inserted anywhere, using parentheses to designate their scope of application.

All styles of composition are supported, including parallel (`[| |]`), interleaving (`|||`), and sequential (`;`). The one flavour of parallel syntax supported at present is interface parallel, where the set of synchronizing events is explicitly listed. Within that set, only bare channel names are permitted. The implication is that any event starting with a listed channel name will be a synchronizing event. The production syntax `{| names |}` is handled properly. Linked parallel and alphabetized parallel composition are not supported.

External choice (`[]`) is supported, but not internal (nondeterministic) choice (`|~|`). An important restriction is that the first event of alternative processes must be explicitly exposed using prefix notation. For example, let:

```
P = a -> A
Q = b -> B
```

Suppose my intention is to choose between `P` and `Q`. Simply writing (`P [] Q`) is not allowed. Instead, I must write (`a -> A [] b -> B`), thereby exposing the initial events of each alternative. This is to make it easy for the translator to identify the events that the choice depends on. In fact, it is equivalent to writing (`a -> A | b -> B`), which is valid CSP but not part of the CSPm dialect. Multiple alternatives can be written as (`a -> A [] b -> B [] c -> C`), and so on.

2.4 Other Constructs

cspt recognizes both single-line (`--`) and block (`{- … -}`) style comments. All declarative statements are ignored: nametype, datatype, subtype, and channel. Presently, these are treated as equivalent to single-line comments, therefore declarations stretching over multiple lines will regrettably result in syntax errors. At the current time, **cspt** does not need to interpret these declarations, but instead infers channel names from operations. Furthermore, all data is assumed to be of integer type. Assert statements (used by FDR2) are also ignored.

In summary, the restrictions detailed above do yield a valid subset of CSPm that can be input to checker, ProBE, and FDR2 without complaint from those tools.

3. Divergence from CSPm

In this section, the features of CSPm that are not fully supported by CSP++ are detailed. They are broken into subsections of unimplemented operators, process parameters, and channel I/O.

3.1 Unimplemented Operators

Some valid CSPm operators not supported, due either to the translator's not handling the syntax, or to the framework's lack of a mechanism to implement the semantics. These are listed in four separate categories to help illuminate their current status and future prospects. The categories are arranged in order of increasing reluctance to tackle them.

3.1.1 Category: Planned for Later

Since data in CSP++ is handled via OO classes and polymorphism, adding support for additional datatypes into the runtime framework is not difficult. Expanded support will be targeted as the need is demonstrated by future case studies. Candidates from CSPm include sets, sequences, and simple enumerated datatypes. Character strings might be introduced as sequences of integer values.

The Boolean guard (`&`) will be added; it is similar to the `if … then` construct already supported.

Implementation of the interrupt (`/\`) operator is planned. It would be very useful, but the framework currently does not contain a mechanism to support it. For example, `P/\Q` would put `P` into a mode whereby prior to executing each event, it would check whether the first event of `Q` has occurred, and if so, terminate itself (as if `P` executed `SKIP`). Regarding UCFs linked from `P`'s events, it will have to be decided whether a blocked UCF should be interruptible, perhaps with some optional cleanup feature.

3.1.2 Category: Low Benefit Cost Ratio

These include constructs that would admittedly be desirable to support, but whose benefits do not presently appear to justify the effort entailed. There are satisfactory workarounds for these cases.

Other flavours of parallel composition, linked and alphabetized, could be added, but interface parallel is already satisfactory. Similarly, the lack of replicated operators can be worked around by writing out all the cases.

`P [] Q` is problematic to translate in the general case. If `P` and `Q` are defined so that their initial events are stated, well and good. But if not, locating the initial events requires considerable manipulation so as to rearrange the process definitions into head normal form [10]. That technique has not yet been pursued in the translator. To some extent, this is a result of the decision to make the C++ output of the translator closely correspond with the CSPm source input.

3.1.3 Category: Questionable in Synthesis Context

Nondeterminism, including internal choice (`|~|`) and "untimed" timeout (`[>`), falls into this category. While nondeterminism can be useful in specifications, it is difficult to think of a clearly appropriate treatment when synthesizing source code.

Some constructs that are not inherently nondeterministic can become such in practice. For example, external choice, where the alternative events are the same, becomes nondeterministic: `e->P [] e->Q`. **cspt** does not detect such cases, and would handle this example by trying event e twice. If event `e` succeeds, `P` will be chosen. If the process has to wait on event `e`, then when `e` eventually occurs, `P` will still be chosen.

3.2 Process Parameters

For now, only integer values are allowed for process parameters. As datatypes are expanded, process parameters will accept non-numeric data. CSPm allows channel names as parameters, and this may also be implemented in CSP++.

3.3 Channel I/O

If any area of CSPm could be described as a quagmire for software synthesis, this is it. The problem of channel I/O, i.e. transferring data from one process to another, is that from the trace semantics viewpoint of CSP, there is honestly no such thing as "I/O," and ProBE and FDR2 reflect this well. To be specific, if a trace is observed to contain the event `foo.1.2.3`, there are many ways it could have got there:

- One process executed `foo.1.2.3`
- Two processes synchronized on `foo.1.2.3`
- One process output `foo!1.2.3`, and another input `foo?x`, or `foo?1.2.y`, or even `foo.1!2?z`
- Two processes synchronized on `foo.1.2.3`, and a third input `foo?x`

Many other combinations are possible, including what could be called "mixed mode" transfers where operators ostensibly calling for output (`!`) appear alongside input (`?`) operators in the same event expression. Furthermore, in interpreting a compound (*dotted*) event, one cannot say by inspection whether some or all components are intended to function as 1- or *n*-dimensional subscripts of the channel name, or whether some or all components are to be considered as data values. It is not difficult to write obscure-looking specifications using these capabilities.

This free-for-all should be contrasted with the original straightforward meaning of "channel" in CSP: A channel was intended to be a primitive structural component in the design of a system, dedicated to one-way, unbuffered, point-to-point data transfer between a particular pair of processes. This kind of definition is extremely easy for system designers to understand and utilize, therefore, it is attractive to implement for the purpose of software synthesis.

The key problem is that channel I/O is, in effect, a metaconcept layered on top of pure event synchronization, and when one looks solely at traces, I/O is found to have dissolved and disappeared. Since ProBE and FDR2 are engaged in state exploration, and since states are represented by traces, it is natural that those tools focus on events, and thus treat I/O in a highly generalized fashion that can barely be recognized as such by programmers. The result is that in ProBE and FDR2, "I/O" operations are treated as pattern matching on events, where "output" (!) asserts components that must match, and "input" (?) designates wildcards that always match, provided any accompanying input datatype constraints are satisfied. After a match has been identified among multiple processes, the full compound event goes into the trace, and any wildcarded components (variables) are bound to copies of the corresponding event components.

From the synthesis standpoint, it was judged that implementing ProBE/FDR2 style pattern matching for events would burden the runtime mechanism with high overhead. Furthermore, it was doubted that such generality was needed or even desirable in practical systems. Instead, CSP++ for the most part reverts to the original meaning of channel I/O, which is a valid subset of CSPm in any case.

The following restrictions have been adopted:

- **cspt** distinguishes between "atomic" events meant only for synchronization, and "channel" events meant for *either* input *or* output.
- The general form of an atomic event is: `chan[.s]*`, where `s` is a numeric subscript and []* represents zero or more instances.
- An output event is: `chan[.s]*!d[.d]*`, where `s` is as above, and `d`'s are data values—numeric expressions or bound variables.
- An input event is: `chan[.s]*?v[.v]*`, where `s` is as above, and `v`'s are unbound variables.
- An output event can transfer multiple data components into a single variable and vice versa. In this skeletal example (which does not work exactly as written), (`cc!1.2.3 || (cc?x -> dd!x) || dd?a.b.c`), `x` would receive `1.2.3`, and then `a`, `b`, and `c` would receive `1`, `2`, and `3`, respectively.
- For synchronization and communication purposes, the channel name and all subscripts must match. The synchronization set for interface parallel composition should contain either the bare (unsubscripted) name of an atomic event {`foo`}, or else the channel name within the closure set (production) notation {`|chan|`}, which will cover all variants of subscripts and data values.

Thus it will be seen that subscripts, if any, must appear before an I/O operator, and that only a single operator, and therefore transfer direction, is allowed. The number of subscripts that appear with a given atomic or channel name must be consistent, or a translation error will result. These restrictions impose considerable clarity on the usage of channels in a specification.

While it may be advisable to use a given channel only for unidirectional communication between a particular pair of processes, the translator does not enforce this. Indeed, broadcast I/O is easy to arrange by means of one outputting process and multiple inputting processes. However, multiple outputters of the same event are not allowed and will result in a runtime error.

4. Extension of CSPm via User-coded Functions

The ability to link CSPm events with UCFs is an essential ingredient of selective formalism. The basic idea is easy to explain.:

When CSP statements are used to model the behaviour of a system, the executions of named events in CSP are intended for two purposes: (1) to synchronize and communicate with other CSP processes; and (2) to mirror what the system *does* in reality. We could say that purpose (1) is for internal use within the specification, but purpose (2) is for external use. Thus, in the classic vending machine example, a `coin.25` event corresponds to the customer inserting a quarter, a `choc` event to pressing the chocolate candy button, and so on. The concept of user-coded functions is essentially to provide some C++ code to bridge the gap between the named CSP events and, in this case, the electronic switch inputs.

Just as two purposes for using events were identified in the previous paragraph, CSP++ makes the restriction that events can be used *either* for internal synchronization and communication, *or* for linking to UCFs. Actually, the step in the design flow where the environment model is removed frees up events that were synchronizing with the simulated environment to be used externally with the real environment. To put it another way, removing the environment model converts the events that were synchronizing it with the implementation model from purpose (1) events into purpose (2) events that are now candidates for linking with UCFs.

At first glance, this restriction may seem purely arbitrary. This question will be revisited below, along with other issues raised by UCFs, after first looking more closely at what UCFs can be used for.

4.1 Nature of UCFs

From the beginning of CSP++ development, it was intended that UCFs be put to practical use in two primary roles, I/O and computation. The first role extends CSPm by providing an interface to external hardware and software. The second role is an escape hatch from CSPm – which was never intended to be a full-featured programming language – allowing programmers to switch into C++ for tasks that would be too awkward to express in CSPm, or too inefficient for execution in translated form.

Under the first role, three flavours of UCFs can be recognized, according to the three types of events that invoke them. This is how their UCFs are invoked by CSP++:

1. *Atomic event:* call UCF, which returns when its processing is “done”
2. *Channel input:* call UCF, which returns when input has been obtained; input data is bound to channel’s variables
3. *Channel output:* call UCF with output values as arguments; UCF returns when output has been accomplished

Case 2 of channel input may involve blocking the process (thread) that is executing the event, but other processes will continue to execute. Timeouts and interrupts are not currently implemented in CSP++, but when they are, this raises the issue of applying them to blocked UCFs.

In case studies to date, this first role has worked well, but plans for UCFs in the second role proved to be too simplistic. The basic problem is illustrated by the following example:

Suppose my e-commerce system needs to calculate the sales tax for a purchase based on the price of the goods and the country they will be shipped too. This calculation would be nicely implemented by looking up the tax rate in a table and doing a multiplication. To represent the lookup table in CSPm would be annoying, and there are no safety or deadlock properties at stake, so this should be a perfect opportunity to drop out of CSPm into a C++ UCF. But how do we write the UCF-linked events in CSPm? The two tools at our disposal are atomic events and channel I/O. The way to make channel I/O work is by visualizing a black-box "ComputeSalesTax" process that has an input channel (for the price and country code) and an output channel (for the tax). Then we might code the following to link to the two UCFs:

```
MARKUP (price, destination) =
  putprice!price.destination -> gettax?tax -> ...
```

The problem here is that the mythical ComputeSalesTax process has to keep track of internal state between the calls to the two UCFs linked to putprice and gettax. In the current version of CSP++, this is left for the programmer to accomplish by means of static storage shared by the two UCFs. This is not very satisfactory, since in the general case the UCFs could be invoked at any time from multiple processes. Probably what is needed is a secure mechanism for the framework to furnish storage to such UCFs on a per-process basis, perhaps by extending the member data of the object that represents the process executing the event.

The above illustrates the case where the UCFs are successively invoked from the same process (i.e. the ends of the channels to and from the "black box" reside in the same CSP process). There is another case, though. Suppose we wish to use UCFs to implement a queue data structure. Then the ends of the enqueue and dequeue channels will very likely be in different CSP processes. What we're proposing here is to replace an entire CSPm process with C++ code. This makes sense under two conditions: 1) the replaced process doesn't need its own thread of control; and 2) it was earlier represented as a CSPm process that was subjected to verification, and we are convinced that the C++ replacement is equivalent. It may be worth building up a library of tested UCFs, for example, of data structures, that are known to be equivalent to given CSPm processes.

4.2 Issues Raised by UCFs

This subsection is organized as a series of four questions and answers.

1. *How can we be sure that UCFs are not breaking the formalism, or giving us a mere veneer of verification?*

Since UCFs are replacing abstract named CSPm events that have no intrinsic meaning, it does not really matter what UCFs do, with one exception: They must not go "behind the back," so to speak, of the CSPm control backbone by engaging in interprocess synchronization or communication. As long as that principle is not violated, any formal properties verified on the CSPm specification should still apply to the synthesized system.

2. *For input-linked UCFs, which party is responsible for validating input, the C++ or the CSPm?*

Validation can be done at either level. As an example, suppose we code the following specification:

```
datatype Num = {1,2,3,4}
channel button : Num
GETINP = button?x:Num -> PROCESS(x)
```

When running ProBE or FDR2, if the environment of GETINP were to offer to engage in button!5, no synchronization would take place. But the **cspt** translator ignores channel declarations and datatypes, so if a UCF were linked to button?x, could it return 5 in x? It could, but it should not. To obey the spirit of CSP, the UCF should validate its input to ensure that it falls in the legal range and is not returned to the control backbone. Alternatively, validation code can be written at the CSPm level, and UCF-linked events can be used to reflect error conditions to the environment.

3. *Events linked to UCFs currently cannot participate directly in choice. Why is that? Can this restriction be overcome?*

The reason for this restriction is that choice is implemented by "trying" an event (i.e. offering to engage in it), and if it succeeds (meaning the offer is accepted), the successor process is executed. If it does not succeed, each alternative is tried in turn. If none are found to succeed, the process is blocked with all alternatives remaining on offer until one is accepted. This kind of try-and-back-out protocol is difficult to coordinate with UCFs, since their current calling sequence is designed to be exercised on a one-shot basis. A more complex calling sequence, which allows direct participation in choice, may be provided in a future version of CSP++. For example, this would be compatible with the programming of polled input.

4. *Events linked to UCFs cannot also be used internally for interprocess synchronization. Can this restriction be overcome?*

It is likely that the main circumstance where this need would arise is when a constraint model is involved. Removing the environment model would normally take away the internal use, but if a constraint model is present, the event may still be needed to synchronize with those processes *as well as* to communicate with the environment.

If we allowed UCF-linked events to also synchronize with other CSPm processes, what would be the implications? To answer this, we must start by identifying the precise time when a UCF involved in synchronization should be called. The only sensible plan is to call the UCF *after* the (two or more) parties arrive at the rendezvous, and of course it must be called exactly once, in order to properly reflect CSP trace semantics. Now let's look at the possible participating events and decide what useful interpretations could be played out:

- *Atomic events:* After recognizing that it is the last party to arrive at the rendezvous, the active party would call the UCF, and then complete synchronization processing (including waking up the other parties).
- *All parties are doing input (?):* This is the broadcast case, from the outside environment to multiple internal processes. The active party would call the UCF and transfer the returned input to all parties, and then complete synchronization processing.

- *Multiple parties are doing output (!):* This is not allowed in CSP++ (see section 3.3 above).
- *One party is doing output, other parties are doing input:* This is also a broadcast case. The active party (who, as the last one to arrive at the rendezvous, knows the output values) would call the UCF to perform the output externally, and then transfer the output to the inputting parties prior to completing synchronization processing.

The above analysis shows that lifting the restriction could be worthwhile. But the programmer would need to understand clearly, on a case by case basis, exactly what a linked UCF was expected to do.

5. Tested Platforms and Performance

By now, CSP++ has been ported to and tested on several different Unix variants, several case studies have been created, and some performance measurements have been taken. These three topics are presented below.

5.1 Platforms

Since CSP++ is currently based on GNU Pth threads, in principle it should be able to run on any platform that Pth supports. So far it has been confirmed to work on Solaris 9 (i86), Redhat Linux 9, Fedora Core 3, and Gentoo Linux, coupled with Pth-2 and the gcc-3 C++ compiler. It is available from the author's website in a zip archive including:

- **cspt** compiler (binary executable)
- CSP++ framework (C++ header files and object library for classes)

5.2 Case Studies

Three case studies have been created. Each one features an initial design made in StateCharts and the derived CSPm statements. In fairness, these are still at the level of "toy" systems, chiefly for proof-of-concept purposes. They demonstrate CSP++ translating and executing the full range of CSPm operators, and the integration of user-coded functions. The references papers all have samples of CSPm and translated C++ code.

- DSS, Disk Server Subsystem—The implementation model includes a disk scheduler and request buffer, with simulated disk driver and simulated clients [7][1][4]. It was originally coded using csp12, but has been recoded in CSPm.
- ATM, Automated Teller Machine—The CSPm includes some verification assertions, and the user-coded functions communicate with a MySQL database [11][12].
- POS, Point-of-Sale Cash Register—This system (in progress) is based on porting CSP++ to uClinux for the Xilinx MicroBlaze embedded processor core implemented on a Virtex-II FPGA [13].

The CSPm and C++ source code for DSS and ATM are available for downloading from the author's website.

5.3 Performance

The DSS case study has been useful for performance metrics, being easy to exercise in a loop (e.g. 20,000 simulated disk requests). In order to make a comparison with a similar-purpose commercial synthesis tool, the DSS system, going back to its StateCharts model, was input to Rational Rose RealTime (RRT, now called Rational Technical Developer). RRT accepts StateCharts as part of a UML model, and generates C++ source code that compiles and links with its own message-driven runtime framework. The comparison is not very ideal, since the operating systems differed (Linux vs. Windows 2000) and also the compilers (g++ vs. Microsoft Visual C++), but tests were performed on the same hardware platforms. The timings (in seconds) are shown in Table 1.

Table 1. Timing for 20,000 Repetitions of DSS

Tool	Run Time	Operating System, Threads	Compiler, Optimization
CSP++ 2.1	1.60 s	Redhat Linux 9, LinuxThreads	gcc 2.96 –O2
RRT	1.47 s	Windows XP	MS VC++ 6.0
CSP++ 4.0	27.03 s	Redhat Linux 9, Pth	gcc 3.2.2 –O2

In measurements with an earlier version 2.1 of CSP++ based on LinuxThreads, the CSP++ implementation of DSS was comparable to the RRT implementation in run time. After porting to Pth, performance deteriorated alarmingly; the cause is under investigation. If Pth is the culprit, another portable thread package will be sought.

6. Related Work

One category of related work is based not on coding in CSP directly, but on providing a library of classes or functions for conventional programming languages that obey CSP's semantics. Rather than promoting direct verification of specifications, this is more an attempt to give software practitioners reliable, well-understood components to build with. Examples of libraries inspired by CSP communication semantics include, for Java, CTJ (formerly called CJT) [14], JCSP [15], and JACK [16]; for C, CCSP [17] and libcsp [18]; and for C++, C++CSP [19] and CTC++ [20]. JCSP and CCSP are a related tool family, as are CJT and CTC++.

Another category features a "straight line" route to verification, like CSP++'s approach, starting with CSP that can be directly verified, and carrying out automatic translation to an executable program. An older tool called CCSP [21] translated a small subset of CSP to C. Recently, the emergence of first-category libraries has facilitated this strategy, and there is now direct translation of CSPm into Java (based on CTJ and JCSP) and C (based on the newer CCSP) [22].

7. Future Work

A good deal of future work has already been implied above in the listing of "divergences." Another potentially fruitful area is performance optimization. Currently, the runtime framework always carries out full environment searching for every event. This allows for dynamic process creation, recursion, and application of renaming and hiding. However, this capability represents overkill for many applications, since CSPm is often used to initially construct a static process structure which is subsequently maintained throughout execution. In that typical system architecture, the translator would be capable of identifying and

binding synchronizing events to one another at translation time, rather than letting the framework search for them over and over again. This would result in significant savings at run time.

CSP++ has always been aimed at embedded systems, but application to real-time systems will require introducing some notion of time. CSP++ is based on the original CSP notation, which does not explicitly model time. While it is already possible to synthesize specifications based on "tock" timing [9], the constant synchronizations on a periodic **tock** event throughout the specification would be grossly inefficient. Instead, it is probably preferable to implement operators from Timed CSP [23]. However, this raises the question of verification, since the Formal Systems tools do not recognize those operators. Adding timed operators to CSP++ would likely suit it for building "soft" real-time systems, but it will probably not be possible to offer the latency guarantees required for "hard" real-time applications.

Further on the theme of targeting embedded systems, porting is underway of CSP++ to an SoPD (system on programmable device) platform [13]. If Pth proves too difficult to port to this platform, there is the option of porting the framework's thread model to a suitable RTOS. This can be accomplished by changing only the `task` class.

Finally, some work has been reported in synthesizing hardware circuits from CSP via Handel-C, an algorithmic hardware description language that has CSP-like constructs [24]. We would like to partition a CSPm specification into software- and hardware-based processes, and synthesize the channel communication between them. This falls under the heading of hardware/software codesign [25]. The aim is to make CSP++ useful for building embedded systems with both hardware and software components, and for SoC (system on chip).

8. Conclusion

To return to the question posed by the title, how faithful is CSP++ to CSPm? The short answer is, faithful enough to be useful. The longer answer is, it doesn't do everything CSPm does, but results suggest that the subset it does do replicates the semantics of CSP. Admittedly, this has not been formally proven.

The development of CSP++ has shown that selective formalism based on software synthesis can be a viable software development technique. Furthermore, the recent commercialization of some CSP-based toolkits indicates that some in industry are seeing practical value to CSP-based approaches. But how can more acceptance of such approaches be achieved? The rest of this conclusion speculates on this topic.

First, we could point out that even without carrying out verification, which admittedly takes training to do well, the CSP++ approach is attractive on its own right. Here are several reasons:

1. Some verification is "automatic" anyway, particularly checking for deadlocks, so if one uses CSPm and FDR2, that will come as a beneficial side effect.
2. Software synthesis is a productivity tool and a way of maturing the software engineering process by putting more emphasis on the specification as the primary design artifact.
3. CSP is a natural, disciplined way to organize the design of concurrent systems, and should make them more reliable, even without verification.
4. CSP is not one of the more obscure formal notations, therefore portions of CSPm specifications can be shown to clients as a way of getting to the bottom of what they really mean by prose requirements.

5. StateCharts are also a nice way to design systems and are useful to show people, and it is easy to convert StateCharts to CSPm for the purpose of software synthesis via CSP++.

Undoubtedly, using CSP *with verification* is much better than without. While the paradigm of selective formalism means that a company would not have to train every software developer in CSP, some CSP gurus would necessarily be required. What human organizational elements are needed to facilitate this?

First of all, it's easy to speculate that sending people for one or two complete university courses in formal methods and CSP is not going to have wide appeal to many managers. Therefore, we would like to find effective ways to bring a typical college-trained programmer up to a level of competency in CSP sufficient to understand, write, and verify CSPm specifications. For this purpose, it is unnecessary to understand deeply the theory of CSP or be able to do proofs. One does have to learn the operators, see and write samples of code according to the "four roles of CSPm" (section 1.2), plug them into ProBE and play with them. In terms of CSP++ specific training, they must learn how to use the synthesis tools and how to link in user-coded functions that obey the restrictions. The concepts behind formal verification are more abstract, but minimal competency using FDR2 is also important. This includes learning how to make simplifications for the sake of verification. Even if the subset of gurus who handle the verification is small, the under-guru level of CSPm practitioners should at least understand what formal verification is about.

From the standpoint of training a cadre of CSPm practitioners, we feel that existing literature on CSP is largely missing a "cookbook" aspect comparable to the popular "Gang of Four" design patterns book [26]. The purpose of that book was to enlighten programmers who already knew the basics of object-oriented programming that "To accomplish common task X, with which you're likely familiar, you code up your classes thusly." This kind of cookbook approach spares programmers from "reinventing the wheel," and, more important, enlightens them on different useful models of "wheels" they would not have imagined for themselves.

Can a similar kind of "CSP design pattern cookbook" be provided for would-be CSPm programmers? This would be a great help in popularizing CSP-based techniques, such as CSP++.

Acknowledgments

This research was supported by NSERC (Natural Science and Engineering Research Council) of Canada.

References

[1] W.B. Gardner, and Micaela Serra. *CSP++: A Framework for Executable Specifications,* chapter 9. In Fayad, M., Schmidt, D., and Johnson, R., editors. *Implementing Application Frameworks: Object-Oriented Frameworks at Work.* John Wiley & Sons. 1999.

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

[3] *FDR2 web site*, Formal Systems (Europe) Limited. `http://www.fsel.com` [as of 5/16/05].

[4] W.B. Gardner. *Bridging CSP and C++ with Selective Formalism and Executable Specifications*, In First ACM & IEEE International Conference on Formal Methods and Models for Co-design (MEMOCODE '03), Mont St-Michel, France, June 2003, pp. 237-245.

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

[6] W.B. Gardner. *Converging CSP Specifications and C++ Programming via Selective Formalism*, ACM Transactions on Embedded Computing Systems (TECS), Vol. 4, No. 2, May 2005, pp. 1-29. Special Issue on Models & Methodologies for Co-Design of Embedded Systems.
[7] W.B. 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.
[8] *GNU Pth – The GNU Portable Threads*. `http://www.gnu.org/software/pth/`.
[9] *Failures-Divergence Refinement: FDR2 User Manual*, May 2, 2003, Formal Systems (Europe) Ltd.
[10] A.W. Roscoe. *The Theory and Practice of Concurrency*. Prentice Hall, 1998.
[11] S. Doxsee, and W.B. Gardner, *Synthesis of C++ Software from Verifiable CSPm Specifications*, to appear in: 12th Annual IEEE International Conference and Workshop on the Engineering of Computer Based Systems (ECBS 2005), Greenbelt, MD, Apr. 4-5, pp. 193-201.
[12] S. Doxsee, and W.B. Gardner, *Synthesis of C++ Software for Automated Teller from CSPm Specifications*, 20th Annual ACM Symposium on Applied Computing (SAC '05), Track: Software Engineering: Applications, Practices, and Tools, poster paper, Santa Fe, NM, Mar. 2005, pp.1565-1566.
[13] J. Carter, M. Xu, and W.B. Gardner, *Rapid Prototyping of Embedded Software Using Selective Formalism*, to appear in: 16th IEEE International Workshop on Rapid System Prototyping (RSP 2005), Montréal, June 8-10, pp. 99-104.
[14] G. Hilderink, J. Broenink, W. Vervoort, and A. Bakkers, *Communicating Java Threads*, Proc. of the 20th World occam and Transputer User Group Technical Meeting, Enschede, The Netherlands, 1997, pp. 48–76.
[15] P.H. Welch, and J.M.R. Martin, *A CSP Model for Java Multithreading*, International Symposium on Software Engineering for Parallel and Distributed Systems (PDSE 2000), Limerick, Ireland, 2000, pp. 114-122.
[16] L. Freitas, A. Cavalcanti, and A. Sampaio, *JACK: A Framework for Process Algebra Implementation in Java*, Proceedings of XVI Simpósio Brasileiro de Engenharia de Software, Sociedade Brasileira de Computacao, Oct. 2002.
[17] J. Moores, *CCSP—A Portable CSP-based Run-time System Supporting C and occam*, in B.M. Cook, editor, Architectures, Languages and Techniques for Concurrent Systems, vol. 57 of Concurrent Systems Engineering series, WoTUG, IOS Press, Amsterdam, the Netherlands, April 1999, pp. 147-168.
[18] R.D. Beton, *`libcsp`—A Building mechanism for CSP Communication and Synchronisation in Multithreaded C Programs*, in P.H. Welch and A.W.P. Bakkers, eds., Communicating Process Architectures 2000, vol. 58 of Concurrent Systems Engineering series, IOS Press, Amsterdam, The Netherlands.
[19] N.C.C. Brown, and P.H. Welch, *An Introduction to the Kent C++CSP Library*, in J.F. Broenink and G.H. Hilderink, eds., Communicating Process Architectures 2003, vol. 61 of Concurrent Systems Engineering Series, IOS Press, Amsterdam, The Netherlands, September 2003, pp. 139-156.
[20] J.F. Broenink, D. Jovanovic and G.H. Hilderink, *Controlling a Mechatronic Setup Using Real-time Linux and CTC++*, S. Stramigioli (Ed.), Proc. Mechatronics 2002, Enschede, The Netherlands, pp. 1323-1331.
[21] 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.
[22] V. Raju, L. Rong, and G.S. Stiles, *Automatic Conversion of CSP to CTJ, JCSP, and CCSP*, Communicating Process Architectures 2003, vol. 61 of Concurrent Systems Engineering Series, IOS Press, 2003.
[23] Steve Schneider, *Concurrent and Real Time Systems: The CSP Approach*, John Wiley & Sons, Inc., New York, NY, 2000.
[24] Jonathan D. Phillips, and G.S. Stiles, *An Automatic Translation of CSP to Handel-C*, Communicating Process Architectures 2004, vol. 62 of Concurrent Systems Engineering Series, IOS Press, pp. 19-37.
[25] Frank Vahid and Tony Givargis. *Embedded System Design: A Unified Hardware/Software Introduction*, John Wiley & Sons, 2002.
[26] Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides. *Design Patterns: Elements of Reusable Object-Oriented Software*. Addison-Wesley, 1995.