Synthesis of C++ Software from Verifiable CSPm Specifications

Stephen Doxsee and W. B. Gardner
Modeling & Design Automation Group, Dept. of Computing & Information Science
University of Guelph, Guelph, Ontario, Canada
sdoxsee@uoguelph.ca, wgardner@cis.uoguelph.ca

Abstract

CSP++ *is an object-oriented application framework for execution of CSP specifications that have been automatically synthesized into C++ source code by the* ***cspt*** *translator. We describe the tool's new capability of accepting input in* ***CSPm*** *syntax, the same dialect processed by the commercial verification tool, FDR2. Using a new ATM case study in CSPm, we give samples of generated code, and illustrate the use of "selective formalism" to code and verify some system functionality in CSP, and supply other functionality via user-coded C++ functions linked to events in the CSP specifications.*

1. Introduction

Formal methods have yet to have any great impact on typical software engineering practices. Although they are effective in the verification of software specifications and contribute towards more reliable software, they are not often taken seriously in industry. This is particularly regrettable in the case of computer based systems that feature concurrent processes in their architecture, because formal models of interprocess synchronization and communication could, if properly applied, eliminate typical pitfalls such as lurking deadlock states.

Personnel trained in formal methods are relatively few, at present, compared to those trained in conventional programming styles, and, in any case, formal methods are outside the comfort zone of most project managers. Therefore, it may be possible to make inroads for formal methods by taking a hybrid approach that does not demand wholesale adoption of arcane notations and mathematical proofs, but instead allows a place for conventional programming. We are advocating such a technique, called *selective formalism*. It is intended to capitalize on both formal methods and traditional software engineering practices by making formal specifications both *executable* and *extensible*. This concept is introduced in the next section, followed by an overview of other attempts to somehow combine formal methods with programming languages.

1.1. Selective formalism

In brief, the notion of selective formalism is to selectively choose to formally specify, at minimum, the critical control portions of a system, and then utilize software synthesis tools to translate the formal specification into executable code. The rest of the system's functionality is provided, as usual, by programming in a popular language, and activating the latter code via the former synthesized control backbone. Thus, selective formalism requires three main ingredients: (i) a suitable formal notation which preferably has verification tool support, and which can be made executable; (ii) a popular programming language; and (iii) some type of framework to tie them together. The synthesis of executable code from formal specifications should be done automatically because of the errors that can be introduced and the time consumed by hand translation.

We chose the process algebra CSP, Communicating Sequential Processes [11][16], as the formally verifiable notation, because of its semantics of interprocess communication and synchronization. CSP is supported by sophisticated commercial tools from Formal Systems (www.fsel.com) such as FDR2, for formal verification, and ProBE, for exploration and simulation of specifications. We chose C++ as the programming language, because it offered an object-oriented approach and is often the programming language of choice for software engineers. The integrating framework for the two is dubbed **CSP++**.

CSP is a textual notation, so some practitioners may feel it lacks in human readability. On the other hand, inputting large system descriptions in graphical form can be slow and tedious, and the readability of diagrams can deteriorate as their complexity rises. So, for example, in the world of hardware design, textual notations such as VHDL and Verilog have largely superceded schematic capture. Furthermore, textual notations are readily compatible with source control systems, and it is easy to track changes on a line-by-line basis.

CSP++ development has been underway for several years [5][6][7][8], and performance measurements on the synthesized C++ code have shown timing on par with a commercial tool that synthesizes C++ from StateCharts (ObjecTime, taken into Rational Rose RealTime, now called Technical Developer) [5][6]. Until now, we faced the significant limitation of translating a local dialect of CSP called csp12, which was not compatible with FDR2. In this paper, we present a new front-end to the CSP++ translator that supports **CSPm** syntax, and demonstrate our framework with a new ATM case study. The upgraded translator allows CSP specifications, verified by FDR2, to be directly translated to C++ without hand massaging.

1.2. Related work

In the quest to stimulate software practitioners to utilize formal methods, there are a number of approaches. A key conflict, which researchers are finding different ways to tackle, is that formal notations are not full-featured programming languages, but the latter are too semantically rich to be amenable to formal verification. Some categories of solutions are listed below, with examples.

The first broad category starts with a programming language that is not immediately formally verifiable, but which can be converted automatically to a verifiable model. This could be called "verification on the side." For example, LOTOS [12] is inspired by CSP and CCS [13] and has a toolset, CADP [4], supporting verification by translating first to a labeled transition system. occam is also inspired by CSP, and [10] gives steps for converting CSP to an occam program. There are variations on this theme.

A second category starts with a conventional informal programming language, but provides a library of classes or functions that obey some formal 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) [9] and JCSP [18], for C, CCSP [14], and for C++, C++CSP [3].

The third category features a "straight line" route to verification, starting with a formal notation that can be directly verified, and carries out automatic translation to an executable program. An older tool called CCSP [1] translated a small subset of CSP to C. Recently, the emergence of second-category libraries has facilitated this approach, and there is now direct translation of CSPm into Java (based on CTJ and JCSP) and C (based on the newer CCSP) [15].

In this spectrum of approaches, CSP++ falls into category three: we start with a formal notation, CSP, which can be verified and automatically translated to C++ (this is "executable specifications"), but we also allow selective user-coded extensions in C++ to be integrated with the formal notation (this is "extensible specifications").

Since many software engineers have never been exposed to CSP, in the next two sections we will take a brief look at CSP and at how CSP specifications can be verified using FDR2. Then we will outline the CSP++ design flow for practicing "selective formalism," briefly describe the workings of the cspt translator and CSP++ execution framework—including a sample of generated source code—and show how to integrate user-coded C++ functions into a CSP control backbone. Finally, a new automated teller (ATM) case study will be used to illustrate the design flow. A Future Work section completes the paper.

2. Overview of CSP

CSP's formal notation contains a small number of fundamental elements: Each statement in a CSP specification is the description of a *process*. The process engages in a sequence of named *events*, which may include point-to-point communication with another process via a nonbuffered, unidirectional *channel*. The set of all events that a process may ever engage in is called its *alphabet*. These may correspond to real-world occurrences such as sensor input, device actuation, and so on. Processes can define themselves in terms of other processes, including several processes running in parallel. Then, the formalism provides for interprocess synchronization each time an event occurs that is in their common alphabet. This also implies that processes synchronize around channel communication.

CSP statements can thus be used to model a system's control and data flow in an intuitive way, constituting a kind of hierarchical behavioral specification. A process definition may terminate with SKIP (normal termination) or STOP (representing a deadlocked system that cannot proceed), or may continue as another process. Here are three simple processes:

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

P carries out the three named events, a, b, and c, then terminates. Q also carries out three events, but then continues as T, performing a fourth event, d, before terminating. The important operators used to create a hierarchical specification allow three flavors of process composition:

(1) Sequential: `P;Q`

Table 1. Translation support for FDR2's CSPm

FDR2's CSPm Features	[15] CSP-to-			CSP++
	CTJ	JCSP	CCSP	
Comments: --	X	X	X	X
Comments: {- ... -}		X	X	X
Integer data	X	X	X	X
Declarations	X	X	X	(1)
Process definitions	X	X	X	X
Recursive processes	X	X	X	X
Parameterized processes: P(2,i)				X
Prefix: ->	X	X	X	X
Chan?data, chan!data	X	X	X	X
Chan?d1.d2. .., chan!d1.d2. ...	X	X	X	X
If ... then ... else ...	X	X	X	X
External choice (alternative): []	X	X	X	X
Interface (sharing) parallel: [|{...}|]	X	X	X	X
Interleaving parallel: P|||Q				X
Sequential composition: P;Q				X
Event renaming: [[e<-f]]				X
Event hiding: \{e}				X
Note (1): not needed for synthesis (treated as one-line comments)				
Not supported				
Boolean guard: & Replicated operators: @ Untimed timeout: [>	Linked and alphabetized parallel Interrupt: /\ Sequences and sets			

(2) Independent concurrent: `P ||| Q`

(3) Synchronized concurrent: `P [|{a}|] Q`

Independent concurrency is also known as process interleaving. In the third expression, the set {a} explicitly declares which events of P's and Q's alphabets they use to synchronize. This means that when P is ready to perform event a, it will be delayed until Q is also ready to perform it, then event a will occur one time (not twice), after which P and Q will each proceed.

A *trace* records the sequence of events that occur from a process execution. Due to the loose execution semantics of CSP, both of these traces would be possible for P||Q synchronizing on a (shown underlined): <r,a,b,c,s,d> <r,a,s,b,c,d>. This usefully matches the way operating systems dynamically schedule tasks on a CPU.

Strictly speaking, channel communication in CSP is just a special case of process synchronization, but it has its own operators to highlight the sense that I/O is being conducted between processes, as in this example:

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

When P and Q synchronize on c, the data 5 is output by P on the channel and input by Q, where it is bound to local variable x. In turn, x can be accessed farther on in Q's definition, and possibly passed to another process as a parameter (a parameterized process is illustrated in Section 6.2.). This channel synchronization is recorded in the trace as <c.5>, and the additional vertical bars in the set expansion notation {|c|} allow for P and Q to synchronize on such compound events that start with the channel name c.

There are a number of other capabilities in CSP, including if/else selection and arithmetic expressions. A key operator is external (or deterministic) choice, which allows a process to interact with its environment and make decisions based on which events the environment is offering to do. In this example, process E stands for the environment:

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

If E offers to do a, then R will continue as P. If E offers b, R will continue as Q.

3. CSPm and FDR2

The original CSP notation proposed by Hoare [11] was not particularly intended for computer processing, hence, machine-readable dialects have arisen. The dialect used for the examples above is CSPm, which is accepted by FDR2 and ProBE. Previously, cspt, the translator for CSP++, only accepted a syntactically different local dialect, called csp12, but now, specifications that are processed by FDR2 can be input directly into cspt for C++ synthesis.

CSP++ currently supports a subset of FDR2's CSPm for software synthesis, but this includes the primary essential operators: process definition (with integer-valued parameters), three styles of process composition (sequential, interleaving, and synchronized), event synchronization, channel data passing, compound events (i.e., having one or more "dotted" components, like foo.18.5.6), event hiding and renaming, and deterministic choice. These are enough for writing even complex specifications and synthesizing C++ software from them. The complete breakdown, with supported features marked by "X", is shown in Table 1, with a comparison to features reported by Raju et al [15] in their translation from CSPm to three libraries (two of Java and one of C). There are no plans for CSP++ to support nondeterministic constructs (chiefly internal choice |~|) for synthesis.

Typically, a CSPm specification can be divided into three sections of definitions:

1) A specification will usually begin with channel declarations and datatype/nametype definitions, for example:

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

The preceding definitions would allow the specification to engage in an output event such as date!12.25.

2) The bulk of the specification is composed of a number of process definitions as already shown above.

3) Finally, assertion definitions specify propositions that will be formally verified. Given P = a -> b -> STOP, we could write:

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

if we wish to verify that the trace <a> is a subset of the traces of P. (This is true since the traces of P are {<>,<a>,<a,b>}). As well as checking for traces refinement as above, FDR2 can also check for failures refinement and failures-divergences refinement to prove properties such as safety and liveness.

With the background in CSP and FDR2 established, we can now proceed to describe the CSP++ design flow.

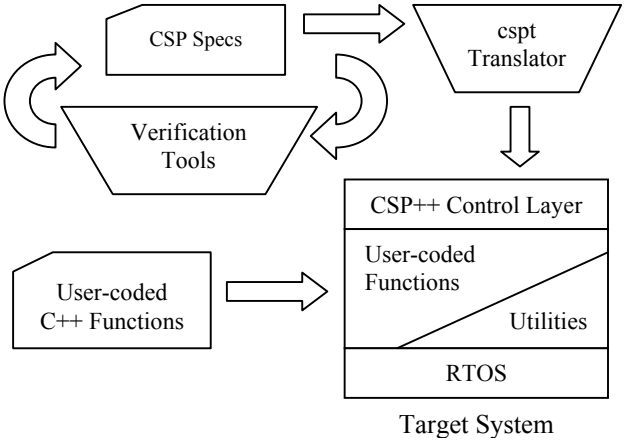

Figure 1. CSP++ design flow

4. The CSP++ Design Flow

The steps of the design flow are shown in Figure 1. The designer starts by creating the CSP specification for the system, and carries out refinement, with the aid of ProBE and FDR2, until satisfied with its behaviour and properties.

To implement the specification in software, the cspt translator is invoked to generate C++ source code. The output file contains CSPm statements interspersed as comments within the generated C++, so it is easy to identify the code resulting from any particular CSP statement. When the C++ is compiled and linked with the CSP++ framework library, the binary can be executed with a command line argument that will cause it to print traces. Examining the execution trace is a means of informally demonstrating that the synthesized software is behaving as expected.

The above steps do not take into account the incorporation of user-coded C++ functions, but this capability is an essential ingredient in "selective formalism." At first glance, it might seem that invoking user-coded functions could break the formalism, but as long as the functions obey certain constraints—chiefly, not to synchronize or communicate "behind the back" of the framework—then in principle the model is maintained.

In practice, the system designer would write CSP specifications for the top level of system functionality—which naturally requires some CSP expertise—and continue writing CSP down to a level where the rest of the coding can be relegated to C++ programmers. How is that level determined, the place where formalism gives way to conventional programming? It is the level below which—

- formal verification is no longer a particular concern because, for example, that part of the system is not safety-critical;
- CSP (which does not pretend to be a full-featured programming language) becomes too awkward for expressing computations and data manipulations which are conveniently coded in C++;
- the abstract named events of the CSP specification logically correspond to events in the system's real environment, and need to be connected to the latter via system calls; or,
- an interface is needed to an external package, such as a database management system.

For any or all of the above reasons, the designer can choose to incorporate user-coded functions, linked to and activated by named CSP events. More will be said about this in Section 5.3. and in the ATM case study below.

5. CSP++ Software Synthesis Framework

The architecture of CSP++ is an object-oriented application framework (OOAF) that incorporates the infrastructure needed to support CSP execution semantics: concurrent threads, event synchronization (including event hiding and renaming), and channel data passing. It takes the form of a C++ class library. The job of the cspt translator is to convert CSPm statements into a system of collaborating objects, whose instantiations and method invocations mirror the original CSP specification. The resulting system can be considered as a customization of the framework. This explanation is expanded in [8].

This translator-plus-OOAF layered approach was adopted, instead of attempting to create a translator that compiles assembly code directly from CSP, for two reasons. First, the OOAF, with its objects designed to mirror CSP entities (processes, channels, events, etc.), was a much easier code generation target than any

processor's assembly language would be. (This is analogous to saying that the Java virtual machine, with its specially designed bytecodes, is an easier target for code generation from Java than assembly language.) Second, the translator and OOAF can be independently modified for porting purposes. In point of fact, the translator has been recently adapted to read CSPm, which required only trivial changes to the OOAF, and similarly the OOAF has been ported to three different threads models, without triggering any changes to the translator.

In the next section, samples are given of the C++ code emitted by the cspt translator for some simple CSPm statements. This is followed by an overview of the structure and operation of the CSP++ OOAF, and the incorporation of user-coded functions.

5.1. Translating CSPm to C++

To illustrate the cspt translator, consider the following simple CSPm specification:

```
channel p,q,r,s,z
A = p->q->z->p->SKIP
B = r->s->z->r->SKIP
SYS = (A [|{z}|] B)
```

This C++ code will be generated for the SYS process:

```
static ActionRef z_r( z );
AGENTPROC( SYS_ )
   z_r.sync();
   {
      Agent::compose( 2 );
      Agent* a1 = START0( A_, 0 );
      Agent* a2 = START0( B_, 1 );
      WAIT( a1 );
      WAIT( a2 );
   }
   Agent::popEnv( 1 );
   END_AGENT;
}
```

The code begins by registering z for synchronization, and anticipating a two-process composition. Then it starts the processes, waits for them to finish, and finally pops the z registration off the environment stack. A and B will run their initial events in no particular order until they reach their respective z events. In CSP's "barrier" style of synchronization, once one process arrives at z, it waits for the other one. Then they perform z together and continue separately to their conclusions (SKIP).

Over the history of CSP++ development, we have implemented concurrent processes using a variety of thread models. Most recently, we have adopted the use of the GNU Pth (www.gnu.org/software/pth). Pth is a portable POSIX/ANSI-C library that provides non-preemptive scheduling.

5.2. The CSP++ framework

As mentioned above, the CSP++ OOAF is essentially a collection of classes whose methods support CSP-style execution semantics. They are designed using memory management techniques that avoid memory leaks. CSP processes are represented by objects of the `Agent` class, and events are instances of either the `Channel` or `Atomic` classes. Each process runs in an environment where certain events have been renamed, hidden, or specified as causing synchronization. Since processes can spawn other processes (via composition) and can, in effect, change their identities (if, for example, the process definition ends in -> *process*), the environment of the starting process (by default, SYS) grows like a tree. CSP++ maintains a branching environment stack at runtime, onto which `EnvRename`, `EnvHide`, and `EnvSync` objects are pushed when their respective CSPm operators are encountered. When a process terminates with SKIP, its branch of the stack is unwound and any environment objects are popped off. The code generated by cspt (some with the aid of preprocessor macros, like AGENTPROC and START0 above) defines temporary objects in local block scope (e.g., `Agent` objects a1 and a2) so that their destructors will be automatically invoked when their corresponding processes terminate. This is one of the memory management strategies referred to above.

One of CSP's powerful features is that of recursive process definitions. In the general case, such specifications could lead to thread and storage explosion at runtime. Fortunately, practical specifications frequently employ a form of tail recursion, for example, P = a -> b -> P, and cspt translates such cases as looping. Similarly, P = x -> y -> Q does not require keeping P's identity alive when execution passes to Q's definition; there is no need to start another thread for Q. But Q must inherit P's environment—the applicable set of synchronizing events, renamings, and event hidings—and the environment stack described above enables that.

In terms of data items, another class hierarchy based on the `Literal` class supports instances of integer-value `Num` objects and list-of-integer `Datum` objects. Support for other CSPm datatypes, such as sequences and sets, can be added in the future by subclassing `Literal`. A container class `Lit` manages dynamically-allocated `Literal` instances throughout their lifetimes—being passed through channels, passed as parameters to processes, bound to local variables—finally deallocating them when no longer referenced. This is another memory management strategy.

Details of the framework's runtime operation, in particular, the means of implementing external choice in the presence of concurrent synchronization, are described in [6] and [7].

5.3. Incorporating user-coded functions

In CSP specifications, it is natural to name events after real-world phenomena: an "ns_sensor" event corresponds to a car standing at a north/south intersection; "red," "yellow," and "green" events represent illuminating traffic signal lamps; and so on. In the specification itself, these are nothing more than abstract symbols, and fodder for verification tools. What selective formalism does, in the context of software synthesis, is provide a way to concretize those symbols by linking them to external devices and/or data processing code. The linkage between a particular event name, say "red," and its C++ function, signalRed, is established when compiling the translated source code, by defining a preprocessor symbol (with suffix _p) like so: g++ … -Dred_p=signalRed.

The designer may use any given event or channel name *either* for internal synchronization purposes, *or* for linking with a user-coded function. In practice, early development of the specification will see event names used to synchronize the CSP functional model (see below) with its simulated environment. Later, as the developer proceeds to software synthesis, the environment model is removed, and the event name is then free to be linked with a user-coded function that actually carries out the input, output, or other processing.

As far as the formal model is concerned, abstract event names are atomic and of indefinite duration, so an event's semantics in the implementation context of the computer based system should be irrelevant. But in order to avoid "breaking" the formalism, it is important that user-coded functions do not perform interprocess communication behind the back, so to speak, of the CSP model. Such violations could possibly reintroduce the concurrency bugs that the use of CSP is meant to avoid, and render the formal verification ineffective.

Next, we turn to a case study that illustrates all the above features.

6. ATM case study

To demonstrate the use of CSP++ with translation of CSPm input, we have implemented a small system, an Automated Teller Machine (ATM). This case study was based on a design by R. Bjork [2], who followed all the steps of OO methodology leading up to a final Java implementation. Of his many UML models—all of which could be considered aids in the implementation of the ATM in CSP++—we have found from our experience that statechart or state machine models can be valuable starting points for coding CSP. One can also write CSP specifications from a series of prose requirements.

State machine representations lend themselves to being modeled in CSP. States become processes and transitions become events. In the case of hierarchical concurrent finite state machines (HCFSMs), as are the state machines of the ATM example, then each state machine can be its own process that synchronizes on common events with other processes. A single large, complex state machine can often be better expressed as HCFSMs [17]. Breaking the design into parts keeps the design understandable.

6.1. CSP in the design phase

CSP notation can be utilized in four roles in the system design phase, constituting four complementary models:

(1) Functional Model: The functional model captures the desired system behavior in terms of CSP processes engaging in named events. For the ATM, UML statecharts were translated to high level CSP to describe the overall behavior of the system.

(2) Environment Model: The environment model simulates 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. In the CSP++ design flow, after implementation the environment model is removed, leaving the system to interact directly with its real environment by means of user-coded functions.

In the case of the ATM, we modeled three other entities that appear in its environment: the CLIENT, the BANK, and the OPERATOR. All of these are simulated, and represent interfaces for the purpose of communication with the ATM. The client process inserts a card, enters a PIN, chooses a transaction, etc., to demonstrate the use cases of the ATM. The bank will process requests made to it (say, for cash withdrawals) and send back responses (e.g., approved or invalid PIN) based on the results of the processing performed. The operator sets the amount of cash in the machine and turns the machine on and off.

(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." In the example of the two traces given in Section 2, <r,a,b,c,s,d> and <r,a,s,b,t,d>, if only a trace where b occurs before s is acceptable, then a constraint process could specify that explicitly. If verification shows that the constraint is violated, the functional model must be improved. The simple ATM system did not require constraints.

(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.

For example, specifying the way to handle invalid PINs was not included in the statecharts, nor in the derived functional model. Such details needed to be specified in CSP so that they could be formally verified.

In the ATM case study, there are four main processes, the BANK, the CLIENT, the OPERATOR, and the ATM itself. The ATM process in turn is composed of several communicating subprocesses. These four are composed in parallel to make up the entire system (SYS). To synthesize the ATM alone, which is the target application, the client, bank, and operator processes would be removed and the channel inputs and outputs of the ATM could then link up with user-coded C++ functions that accept button pushes, provide network connections, etc.

6.2. Code samples

The following provides a sample of some of the CSPm code for the ATM. The entire case study is available on the Web; see Section 8. The first item of interest is a method of providing, in effect, a global variable for the PIN. It uses get and set channels to access the PIN value. PINi is its initial state, and PIN(val) is a parameterized process.

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

Here is the specification for a single session, which was derived from a statecharts model.

```
SESSION = insertcomplete -> READINGCARD

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

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

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

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

EJECT = ejectcard -> DONE

DONE = sessiondone -> SESSION
```

Also in the ATM is the TRANSACTION processor, shown with one parameterized subprocess, SPECIFICS(1), that handles a withdrawal:

```
TRANSACTION = chosen?menu -> SPECIFICS(menu)

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

SEND and RECEIVE are responsible for communication with the BANK process:

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

RECEIVE(menu) = bankstatus?stat.pin.val ->
      (if (stat == 1) then invalidPIN ->
         HANDLEPIN(menu,pin,val,1)
       else if (stat == 2) then approved ->
         COMPLETING(menu,val)
       else rollback -> ANOTHER)
```

Finally, the ATM process is composed of SESSION, TRANSACTION, VARIABLES (including PINi), and OVERALL (not shown above):

```
ATM =
( (OVERALL [|{|insertcomplete, sessiondone|}|]
   SESSION)
[|{|choose,endtrans,anothertrans,holdingcard|}|]
  TRANSACTION)
[|{|cardset,cardget,pinset,pinget,machset,
  machget,balset,balget|}|] VARIABLES
```

6.3. Verification and functional test cases

There are some system properties that FDR2 can check on its own with no special instructions, as indicated by these assertions:

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

Realistically, most useful verification involves learning how to “ask the right questions” of the tool.

The ATM design documents provided functional test cases that proved useful for coding assertions. By using the trace refinement method in our case study, we are able to prove one of the functional test cases and show that a client's card will indeed be held after entering the wrong PIN three times in a row:

If there are three invalidPIN events in the same transaction, the card will be held and no receipt (indicating a completed transaction) will be issued. Using trace refinement, mentioned in Section 3, we can analyze the following two assertions that prove the aforementioned test cases. (The ‘\’ operator is used to hide events that are not germane to the assertion.)

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

The assertion should succeed since the card must

indeed be held after three invalidPIN events. Now if we change the trace portion of the assertion to this:

```
[T= invalidPIN -> invalidPIN -> invalidPIN ->
   receipt -> STOP
```

it should fail, since there is no trace that can have three invalidPIN events followed by the issuing of a receipt.

An example of a liveness specification comes from the System Startup use case: we want to prove that the ATM must continue to allow activation of the "on" switch, a request for the initial cash amount, followed by activation of the "off" switch. That is, the failures of the ATM process should be a subset of failures of a specification that repeatedly performs the sequence of on, machcash.*x*, and off. In a liveness specification "P [F= Q", P puts a limit on what Q can do and requires Q to accept at least a certain range of behaviours. We write this as follows:

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

FDR2 confirms that this liveness specification is satisfied. Other properties can be proved in a similar fashion. For example, we can verify how the ATM will respond if the bank sends back a message stating that the amount requested exceeds the account balance.

A potential problem using FDR2 for verification is state space explosion. When we verified the ATM specification, values such as account balances were limited to a few possible values in order to prevent state explosion. But since cspt does not need to analyze the state space to produce its translation, this is not a problem for the CSP++ implementation. It should be noted, though, that specifications can be synthesized that result in resource exhaustion at runtime, just as conventional programs can cause stack overflow through careless use of recursion.

6.4. User-coded functions for the ATM

The following is a representative case of linking the channel event 'readpin?p' of the READINGPIN process with a user-coded function that obtains the input PIN and returns it to the CSP specification via the status argument.

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

In the generated C++ code below, assume that it was compiled with "-Dreadpin_p=readpin_chanInput":

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

Now when the Channel object readpin has its extraction (>>) operator invoked, the external linkage to the function readpin_chanInput will be used, instead of attempting to synchronize with a channel output operation in another process.

Other events in the ATM specification are candidates for linking to user-coded functions. For example, the banksend and bankstatus channels of the SEND and RECEIVE processes (Section 6.2. above) were linked to functions that make network socket connections with a bank system that processes transactions and maintains an SQL database of client account information.

6.5. Debugging

Finding and eliminating defects takes place in different phases of system development. At the CSP specification stage, Formal Systems' Checker tool can help diagnose erroneous use of datatypes, and ProBE can be used to check the syntax and explore the state space of the specification. After cspt synthesizes C++, the compiled code can be executed with trace printing turned on. When execution ceases—either because there are no unblocked processes to run, or because a process executed STOP—a process status dump is printed, showing what events each process is waiting on. The dump also records the "high water marks" of the numbers of concurrent threads created and literals allocated. For low level checking, the OOAF can be recompiled with the ACTWATCH preprocessor symbol that causes it to print a running log of all choice-making and synchronization activities. Another symbol, MEMWATCH, activates logging of all literal allocations and deallocations. The compiled code can be executed under control of GNU gdb, allowing for breakpoints, single-stepping, and storage inspection in both the synthesized code and in the user-coded functions.

7. Future work

The key objective of enhancing CSP++'s usability has been recently accomplished by realigning the translator's front end with FDR2's CSPm syntax. Work that is underway or planned include the following areas:

1. Two remaining restrictions result from the former

csp12 dialect's semantics. (1) An event name used to synchronize two processes cannot be reused to synchronize those two with another process at a higher level of composition. The desired behaviour is for all three processes to synchronize on the same event, which in csp12 was obtained by writing (P||Q||R)^{*sync set*}. CSPm specifications that need this behavior must work around the restriction for now. (2) When the external choice operator [] is used, the first event of each process must be explicitly exposed. Thus, a->P[]b->Q is synthesizable, but P[]Q is not. In csp12, choice was written as a->P|b->Q.

2. Support for more CSPm datatypes will be added.
3. CSP++ implements the original definition of CSP that does not include timing and interrupts. Until support is added, specifications written in terms of event-based "tock" timing [16] can be synthesized (though with awkwardness due to restriction (1)).

In the future, it is intended to extend synthesis capabilities to hardware description language for application to hardware/software codesign.

8. Conclusion

In this paper, we presented a new front-end to the CSP++ translator that supports CSPm syntax and demonstrated by means of an ATM case study that selective formalism can be a worthwhile software process model for the design and development of computer based concurrent systems. We believe that selective formalism is an attractive approach to concurrent system design and development because of its flexible combination of formal verification and conventional programming, all based on executable and extensible specifications. The latest version of CSP++ can be downloaded from the author's website http://www.cis.uoguelph.ca/~wgardner. The code for the ATM case study is included.

9. References

[1] 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.

[2] R.C. Bjork, An Example of Object-Oriented Design: An ATM Simulation. http://www.math-cs.gordon.edu/local/courses/cs211/ATMExample/.

[3] 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*, volume 61 of *Concurrent Systems Engineering Series*, IOS Press, Amsterdam, The Netherlands, September 2003, pp. 139-156.

[4] J. Fernandez, H. Garavel, L. Mounier, A. Rasse, C. Rodriguez, and J. Sifakis, "A Toolbox for the Verification of LOTOS Programs," *Proc. of the 14th International Conference on Software Engineering* (ICSE'14), Melbourne, May 1992, pp. 246-259.

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

[6] W.B. Gardner, "CSP++: An Object-Oriented Application Framework for Software Synthesis from CSP Specifications," Ph. D. dissertation, Department of Computer Science, University of Victoria, Canada. 2000. http://www.cis.uoguelph.ca/~wgardner/, Research link.

[7] W.B. Gardner, "Converging CSP Specifications and C++ Programming via Selective Formalism," to appear in *ACM Transactions on Embedded Computing Systems* (TECS), Special Issue on Models & Methodologies for Co-Design of Embedded Systems.

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

[9] 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.

[10] M.G. Hinchey, and S.A. Jarvis, *Concurrent Systems: Formal Development in CSP*, McGraw-Hill Book Company. 1995.

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

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

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

[14] 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,* WoTUG, IOS Press, Amsterdam, the Netherlands, vol. 57 of Concurrent Systems Engineering series, April 1999, pp. 147-168.

[15] V. Raju, L. Rong, and G.S. Stiles, "Automatic Conversion of CSP to CTJ, JCSP, and CCSP," *Communicating Process Architectures 2003,* IOS Press, 2003.

[16] Steve Schneider, *Concurrent and Real Time Systems: The CSP Approach*, John Wiley & Sons, Inc., New York, NY, 2000.

[17] Frank Vahid and Tony Givargis, *Embedded System Design: A Unified Hardware/Software Introduction*, John Wiley & Sons, 2002.

[18] 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.