Cspt: An Open Source Translator for CSPm

W.B. Gardner
Department of Computing and Information Science, University of Guelph, ON, Canada
gardnerw@uoguelph.ca

Abstract

A translator for CSP specifications coded in CSPm is released as open source. It generates C++ source code that instantiates classes of the CSP++ framework, which in turn provides CSP execution semantics. The architecture of the translator is described to give developers a starting point for modifications, e.g., adding operators, replacing the back end with another code generator, or building an interpreter, verification tools, etc.

1. Introduction

Many practitioners utilize Communicating Sequential Processes (CSP) [1] notation to write specifications for concurrent systems. This motivates the desire to build tools that can interpret a machine-readable dialect of CSP, whether the objective is algebraic manipulation of specifications, automated verification, or system synthesis. In any case, parsing a file of CSP text is a primary requirement.

The **CSP++** tool chain [2] contains such a parser, which is now available as open source. CSP++ provides the means to make specifications written in the CSPm dialect executable on Unix-variant platforms. This paper describes the architecture of its translator, **cspt**, for the benefit of researchers and developers who wish to modify it or adapt it to their diverse purposes.

The original tool chain consists of two components: (1) the runtime library, which is supplied in the form of an object-oriented application framework (OOAF); and (2) the translator that produces a "customization" of the OOAF by outputting C++ source code which instantiates framework classes. Compiling the C++ source code and linking with the framework's library results in a program that produces the effect of executing the CSP specification. Through a concept dubbed "selective formalism," individual CSP events and channels may be linked to user coded functions (UCFs) written in C++. UCFs may be used for purposes such as computation (since CSP is not intended to be a full-fledged programming language) that is not judged necessary to be formally specified, or performing I/O. UCFs must not engage in interprocess communication or synchronization, since that could break the formal model. The idea of software synthesis from a formal specification, coupled with selective formalism, builds a bridge, in effect, between the abstraction level of formal methods and the detailed implementation of conventional programming [3].

Heretofore, the translator has been distributed only in binary form for Solaris (x86) and Linux platforms. It is now available from www.cis.uoguelph.ca/~wgardner, "Research and Downloads" page, as C++ source code under the GNU General Public License (GPL version 2). The intention and requirement of the GPL is that developers who leverage the existing code base to create new tools for CSP will, in turn, make their derivative works publicly available as source code under the GPL, thus widening the benefits to the user community.

The following sections describe the subset of CSPm accepted by version 4.2 of cspt, and then give an overview and description of its processing phases and important data structures, without which simply reading the code would be a weary exercise. The output of cspt, i.e., the C++ generated for each CSPm construct, is not covered here (see [4, 5]). Likewise, the execution semantics of CSP++ are discussed most fully in [6]. The paper closes with some suggestions for future enhancements.

2. CSPm input dialect

It should first be noted that cspt was originally created to accept the dialect of CSP used by **csp12** [7], an in-house verification tool written in Prolog. This dialect was incompatible with that used by the popular commercial verifications tools, FDR2, ProBE, and checker, from Formal Systems Europe Limited, and the decision was made in version 4.0 of CSP++ [8, 9] to change over to CSPm completely. As a result of this history, the nomenclature of csp12 and support for some of its operators remain in the code base, which is apt to confuse readers unmindful of the background.

References for CSPm syntax are Scattergood's the-

sis [10] and Appendix A of the FDR2 user manual [11] also found in Appendix B of [12]. Cspt translates a large and useful subset of CSPm, as shown in these categories (see [6] for details):

1. *Identical to FDR2:*
   - *comments:* line `--`, block `{- -}`
   - *operators:* conditional (`if...then...else`), prefix `->`, renaming `[[<-]]`, hiding `\{}`, input `?`, output `!`, arithmetic and relational expressions
   - *composition:* interface parallel `[|{...}|]`, interleaving `|||`, sequential `;`
2. *Recognized, but treated as single-line comments:* `channel`, `datatype`, `nametype`, `subtype`, `assert` statements
3. *Limited support:* external choice (must expose first events: `a->P[]b->Q`), compound (dotted) channel data (see below), process parameters (only integers), synchronization set closure notation `{|...|}` (only channel names)
4. *Not recognized:* mixed mode channel I/O (pattern matching, e.g., `chan?x!x+2`), in-line datatype for channel input (`chan?x:Type`), internal (nondeterministic) choice `|~|`, other kinds of parallel composition, replicated operators, "untimed timeout" `[>` and interrupt `/\`, and all other keywords

Since channel statements are mandatory for FDR2, category 2 treatment allows verified source files to be sent to cspt without editing. Cspt infers channel and event names from operations, and supports only integer data, so it has no need for these statements.

An ambiguous feature of CSPm notation is that, given a "dotted" event like `foo.3.27.3` in a system trace, one cannot say whether the author intended this to mean "communicate tuple <3,27,3> on channel foo," or "communicate <27,3> on channel foo_3." Compound channel data is handled by cspt using the rule that only one I/O operator, ? or !, is allowed; dotted components (if any) to the left of the operator are taken as channel subscripts, and those to its right as compound data (tuples). This restriction is compatible with CSPm, while avoiding ambiguity in implementation.

3. Overview of translator

From this point on in the paper, the terminology shifts from CSPm in order to match the software. Table 1 gives the corresponding terms.

Table 1. CSPm versus csp12 nomenclature

CSPm	csp12 and cspt software
process	agent
event, channel	action, in two types: atomic (synchronization) channel (communication)

Cspt operates in two phases: first, a combined lexical and syntax phase based on **flex** and **bison**, which scans the CSPm input file (.csp and .fdr2 extensions are recognized) and produces a syntax tree; second, a code generation phase that walks the tree and produces a C++ output file. In addition to the tree, the symbol tables persist between phases. The main goal of translation is to render each CSPm process ("agent") definition as one or more C++ functions (known as AgentProcs). Control will pass to these functions at run time using a user-level threading mechanism.

Cspt has been equipped with the following command line switches:

- -d: debug mode (prints parse tree)
- -s: interleave CSPm in C++ output as comments
- -t: turn bison tracing on (yydebug)
- -w: place C++ output in working directory (instead of the same directory as CSPm specification)

Object-oriented design has been used throughout. The syntax tree is built from `ParseNode` objects (explained below), each of which knows how to generate itself and its subtree during phase two. The C++ Standard Template Library (STL) has been utilized for stock data structures, especially in the areas of tree building, tree-navigation, and symbol table access.

Some attention has been given to diagnosing problems in the translator's input, down to the offending line and character where practical. Still, this feature is fairly rudimentary at present.

Four technical issues will arise in the description that follows:

1. *Identifying and extracting complex subagents for separate generation, while ensuring access to their parents' symbols*

Take, for example, the simple definition:

```
SYS = c?x-> (SERVER [|{|request,result|}|]
       (CLIENT(0,x) ||| CLIENT(1,x)))
```

This cannot be translated into a single function. Instead, a "subagent" will be extracted to implement `CLIENT(0,x) ||| CLIENT(1,x)`, and then `SYS` can

start the SERVER thread and the subagent's thread in parallel. Variable `x` in `SYS` must be accessible to the subagent, so it can be passed to each `CLIENT`.

2. *Managing the symbols for multiple agent definitions and binding them to agent invocations*

In the example above, there are two different invocations of CLIENT. Suppose definitions for `CLIENT(0,0)` and `CLIENT(i,n)` were supplied in the CSPm specification. In other words, $CLIENT_{0,0}$ is a special case. (Special cases must be listed first in order to work properly.) Given `CLIENT(1,x)`, the translator can unambiguously bind this invocation to `CLIENT(i,n)`. However, `CLIENT(0,x)` must be bound at run time, when the value of `x` is known.

3. *Handling agent invocation and termination, depending on the context*

In the definition `S=P;Q`, different treatments are given to the starting of P and Q. S will start P as a new thread and wait for it to finish. (It cannot "call" P because CSP++ does not have a call/return stack mechanism at the AgentProc level.) But it can merely branch ("chain") to Q, since there is no requirement for control to return to S.

4. *Linking to UCFs*

If the programmer supplies, in a separately compiled C++ file, a UCF called "fred" to be invoked for channel input `fred?x`, this linkage is arranged at compile time by defining the preprocessor symbol `fred_p=fred`. This overrides the default (null) definition of `fred_p` in the generated code. Definitions that are null at run time result in synchronization within the specification instead of calling UCFs.

4. Lexical and syntax phase

Translation is driven by the bison-generated parser, which is invoked from the `main()` function in file cspm.y. The parser shifts input tokens onto its stack until it recognizes patterns in the language's grammar. These are supplied in pseudo-BNF style when the bison generator is invoked. The parser, in turn, gets its input tokens from the flex-generated scanner, which breaks them out of the input stream according to the language's lexical features. A routine YY_INPUT is supplied to read the input file and fill the scanner's buffer. When the parser recognizes a grammar pattern, it invokes methods that add nodes to the parse tree.

The rules used to customize flex and bison are given below, followed by a description of the relevant data structures, the parse tree and the symbol tables.

4.1. Lexical rules

These rules are contained in the flex source file cspm.lex. The scanner is told to recognize and skip over CSPm style comments. Then, the CSPm operators and delimiters are listed, ranging from single-character tokens such as "!" to double- and triple-character tokens "->", "|||", etc. A handful of reserved words are recognized by flex such as `if`, `SKIP`, and `STOP`. Such tokens are reported to the parser by number. For single-character tokens, the ASCII value of the character is its number. The others are assigned code numbers from a table in the file cspm.tab.h, which is generated by bison from the `%token` statements appearing in file cspm.y.

The above description applies to tokens that are fixed character strings. Other kinds of tokens, such as numbers and identifiers, are recognized by regular expression patterns. CSPm uses two types, `ID` and `NUM`. `NUM` tokens are reported to the parser using their integer value. `ID` tokens start with a letter and can contain any alphanumeric character, plus "_" and "'". They are stored as C++ `string` objects, the instances being allocated by the scanner. The value of an `ID` passed to the parser is its `string*` pointer.

4.2. Grammar rules

These rules are contained in the bison file cspm.y. They are listed in Table 2, rewritten in conventional BNF. Thus, Table 2 documents both the accepted input syntax and the translator's processing, side-by-side in one place. It should be the starting place for anyone wishing to modify the translator.

4.3. Parse tree

Bison rule recognition results in either creating a new `ParseNode`, or adding a token to an operand list in preparation for creating a `ParseNode` from the list. An important typedef is `LOPNP`, an acronym for "list of `ParseNode` pointers." `LOPNP` is defined as the STL template `deque<ParseNode*>`, and manipulated with the STL's container class methods.

The `ParseNode` class hierarchy (ParseNode.h) is shown in Figure 1 using UML. The abstract base class records the CSPm input file line number associated with the node (used to print diagnostics for errors discovered in phase two). The virtual `prep()` and `gen()` methods (explained in Section 5) specify the

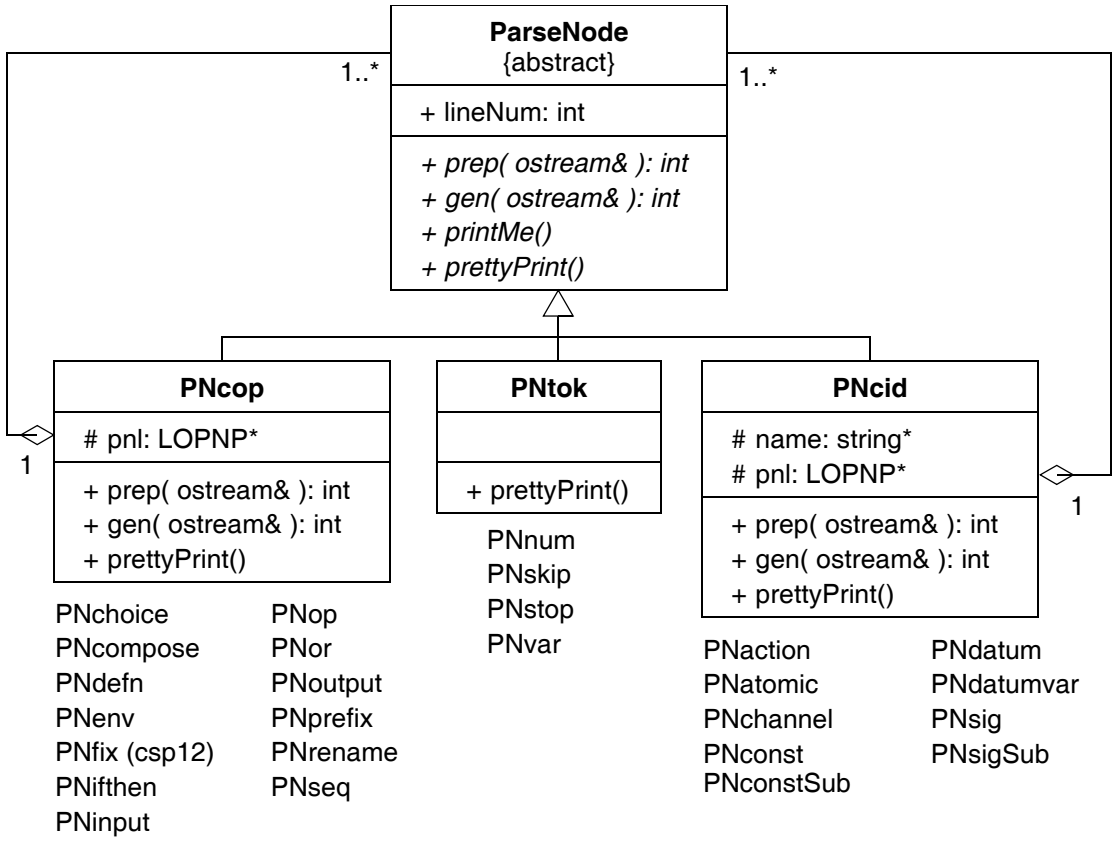

Figure 1. Parse node class hierarchy (V4.2)

output stream to print to, and provide a status return which can be used to abort the translation.

The subclasses define three broad categories of specific parse node types:

- `PNcop`: complex operators, having a list of operands (`pnl`)
- `PNtok`: simple tokens
- `PNcid`: complex identifiers, having a name and a list of arguments or subscripts

These all have default `prep()` and `gen()` methods, which may or may not suit their subclasses. If not, the methods can be overridden.

The specialized parse node subclasses (ParseNodes.h, note plural) are listed in Figure 1 underneath their respective base classes. Inheritance goes down just one level, with the exception of `PNconst`, which is the parent of `PNconstSub`. These classes are listed opposite their corresponding BNF statements in Table 2. Their constructors (abbreviated "ctor" in the table), in most cases simply store their arguments, though some perform symbol table lookups (see next section).

For debugging purposes, virtual methods are provided for printing each parsed definition. When the "-d" debug flag is used on the cspt command line, `prettyPrint()` is called each time an agent definition is recognized. Complex nodes print their name or type, and then recursively `prettyPrint()` each of their operands. Simple nodes print their name or value. The result is a neatly indented, nested representation of the parse tree printed on `cerr` (`stderr`).

Not shown in Figure 1 are a handful of additional virtual methods used for querying node types and values: `isNum()`, `intVal()`, `getName()`, and the like. The abstract base class supplies default methods for these (always returning false or 0); relevant subclasses simply override these defaults.

4.4. Symbol tables

Symbol collection is an important task of the translator's first phase. A number of `ParseNode` subclass constructors require access to symbol tables: storing new names and looking up existing ones.

Symbol tables are constructed with the help of the STL `map` template. The `SymTable<T>` class (Symbols.h), derived from `map<string*, T*, SYcompare>`, sets up a mapping from identifiers (`string*`) to symbol entries (`T*`). The "compare object" `SYcompare` provides the operator needed to order any pair of `string*` identifiers by invoking the

Table 2. BNF syntax with corresponding parse node classes (V4.2)

Accepted CSPm syntax in BNF	Parent			Subclass Name	Pseudocode[a]		
	PNtok	PNcop	PNcid		ctor[b]	prep()	gen() and details for entries marked ">" (ctor = constructor)
	ParseNode				>	OK	ctor: store line number gen(): OK
	*	PNtok			{ }	-	-
		*	PNcop		{ }	apply prep/gen to each operand in turn; stop on bad status	
			*	PNcid	{ }	>	prep(): apply to each arg/subscript; stop on bad status gen(): output name
<definition> ::= <signature> '=' <agent>		*		PNdefn	{ }	NC	prep signature and agent; use agent's symbol entry to gen AGENTPROC, arg #defines, and FreeVars (genAgentProc); gen agent body; "ENDAGENT" if needed; gen arg #undefs (genEndAgent)
| <agent> '[]' <agent> { '[]' <agent> }		*		PNchoice	>	-	ctor: continue only if all agents are prefix"Agent::startDChoice(n)"; set flag for PNinput (DatumVar gen); genPre actions;"Agent::whichDChoice()"; genPost agents
| <agent> ';' <agent> {';' <agent>}		*		PNseq	{ }	-	gen each agent, flagging last one
| <agent> '||' <agent> | <agent> '|||' <agent>		*		PNcompose	{ }	>	prep(): prep simple agents; complex: use agent's symbol entry to extract subagents (makeSubAgent), then gen gen(): "Agent::compose(*n*)"; "START" each agent; "WAIT" each agent
| <agent> '^' '{' <name>{,<name>} '}' | <agent> '\' '{' <name>{,<name>} '}' | <agent> '#' '{' <rename>{,<rename>} '}'		*		PNenv	{ }	-	gen the ActionRefs; "."; "sync()", "hide()", or gen PNrename; gen the associated agent; "Agent::popEnv(*n*)" if needed
| <agent> '+' <agent>		*		PNor	{ }	-	gen each agent
| STOP	*			PNstop	{ }	-	"Agent::stop()"
| SKIP	*			PNskip	{ }	-	set flag to get ENDAGENT generated

Table 2. BNF syntax with corresponding parse node classes (V4.2) (Cont.)

Accepted CSPm syntax in BNF	Parent			Subclass Name	Pseudocode[a]		
	PNtok	PNcop	PNcid		ctor[b]	prep()	gen() and details for entries marked ">" (ctor = constructor)
| <UID>['(' <exp>{,<exp>} ')']			*	PNconst	{ }	>	prep(): find in agentTable, get agentproc name via bindSig(*args*) gen(): "CHAIN", "START", or "START/WAIT" depending on context
| IF <exp> THEN <agent>)		*		PNifthen	{ }	>	prep(): prep agent gen(): "if ("; gen exp; ") {"; gen agent; "}"
<prefix> ::= <action> '->' <agent>		*		PNprefix	{ }	-	gen(): - genPre(): gen action genPost(): gen agent
<signature> ::= <UID> '(' <numvar>{,<numvar>} ')'			*	PNsig	>	>	ctor: find in agentTable, or insert new variant prep(): find signature in agentTable, set its symbol entry as the translation context; setup symbol entry to handle symbols for variant (prep) gen(): NC
<numvar> ::= (<NUM>	*			PNnum	{ }	-	output value
| <VAR>)	*			PNvar	{ }	>	prep(): report to agent's symbol entry (addvar) with "global" flag if in subagent gen(): output var name, maybe globalized, obtained from agent's symbol entry (ref)
<action> ::= (DONE	*			PNdone	{ }	-	-
| <LID>['(' <exp>{,<exp>} ')']			*	PNatomic	>	OK	ctor: find/insert in actionTable gen(): output name, gen subscripts
| <LID> '?' (<VAR> | <datumvar>)		*		PNinput	>	-	ctor: new PNchannel gen(): if datumvar, "DatumVar" *temp* "=" gen datumvar; gen PNchannel; ">>"; gen PNvar or *temp*

Table 2. BNF syntax with corresponding parse node classes (V4.2) (Cont.)

Accepted CSPm syntax in BNF	Parent			Subclass Name	Pseudocode[a]		
	PNtok	PNcop	PNcid		ctor[b]	prep()	gen() and details for entries marked ">" (ctor = constructor)
| <LID> '!' <exp>)		*		PNoutput	>	OK	ctor: new PNchannel gen(): gen PNchannel; "<< ("; gen exp; ")"
	*			PNchannel	>	-	ctor: find/insert in actionTable gen(): output name
<datumvar> ::= <LID>['(' <VAR>{,<VAR>} ')']			*	PNdatumvar	>	-	ctor: find/insert in datumTable gen(): output name, gen subscripts
<name> ::= <LID>['(' <NUM>{,<NUM>} ')']			*	PNaction	{ }	OK	find in actionTable, output ActionRef, gen subscripts
<rename> ::= <name> '=' <name>		*		PNrename	>	OK	ctor: get 2nd name into actionTable (makeAtomic) gen(): gen 1st PNaction; ".rename("; gen 2nd PNaction; ")"
<exp> ::= (<numvar>	*see <numvar> above*						
| <LID> '(' <exp>{,<exp>} ')'			*	PNdatum	>	OK	ctor: find/insert in datumTable gen(): output name, gen subscripts
| '–' <exp> | <exp> <op> <exp>) <op> ::= ('+' | '–' | '*' | '/' | '=' | '<' | '>' | '=<' | '=>' | '<>')		*		PNop	{ }	OK	"("; gen left exp; *op*; gen right exp; ")"
Prep-time node substitution: new extracted subagent's <signature>			*	PNsigSub	{ }	>	prep(): note subagent no. in translation context gen(): NC
replaces complex <agent> subtree, refers to subagent	PNconst			PNconstSub	{ }	OK	default to PNconst::gen()

a. *Pseudocode abbreviations:* { } = no-op; - = default to parent's method; OK = no-op, return good status (0); NC = method is not called; "foo" = output "foo"

b. *Constructor note:* The obvious action of storing arguments in data members is not explicitly written out.

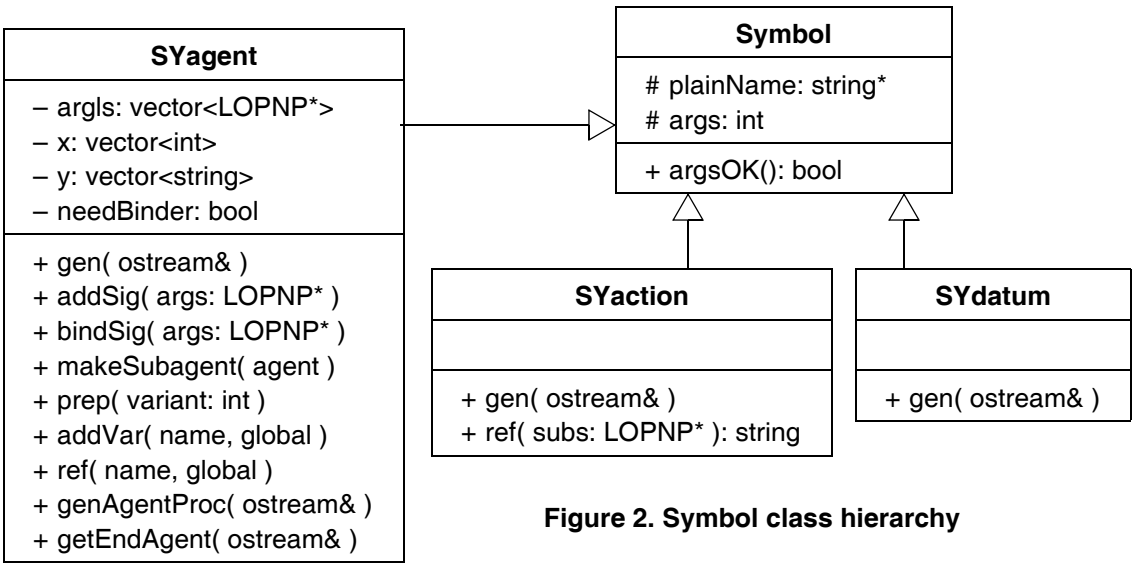

Figure 2. Symbol class hierarchy

C++ `string::compare()` function. The underlying `map` functionality ensures that there are no duplicate definitions in a `SymTable` object, and allows its symbols to be iterated in lexicographic order.

Different kinds of symbol entries are defined for agents, actions, and datums. In csp12, *datums* were a kind of complex data object capable of being output over a channel and stored in a datum variable (`DatumVar`). A datum had a name followed by components in parentheses, which could be integers or nested datums, e.g., `request(12,3,block(2))`. For CSPm, datums are still utilized internally, with dummy names and without nesting, as a way of packaging tuples of channel data.

The hierarchy for the symbol table entry classes is shown in Figure 2. The only method these three tables, or more precisely, the symbol entry classes, have in common is the `gen()` method, used to output symbol definitions in the code generation phase. Other features of these classes are described next.

4.4.1. SYagent entry. One `SYagent` object is created to record all *variants* of a particular agent name; e.g., `CPU(1,1)`, `CPU(2,i)`, and so on, are variants of the name "CPU". Once an agent name has been defined, it sets the pattern regarding arguments, and all subsequent variants must have the same argument cardinality. That is, a subsequent definition of `CPU(n)` would be rejected. This is a restriction of CSP++.

As each variant is encountered in the input stream, it is processed into the same-named symbol entry by the `addSig(`*args*`)` method:

- Its argument list is appended to the `argls` vector.
- The arguments are analyzed, resulting in its signature and appropriate entries being appended to the `y` and `x` vectors, respectively.

These vectors are later used for agent binding via the `bindSig(`*args*`)` method. If translation-time binding ever fails, the `needBinder` flag gets set, which will cause an `AgentBinder` object to be generated when `gen()` is later invoked on the symbol entry. This object provides the data needed so the runtime binding mechanism can choose the correct variant for a given invocation of this agent.

When the code generation phase commences, subagents may be extracted. In that case a special `PNdefn` node needs to be created containing the extracted subtree. This chore is performed by the `makeSubagent(`*agent*`)` method.

The *agent* argument (that is, the subtree) gets replaced by a `PNconstSub` node referring to the subagent name. Such names are assigned sequential numbers within the symbol entry (e.g., `SYS_s1`, `SYS_s2`, etc.). The subtree itself is reinstalled under a new `PNdefn` node, along with a `PNsigSub` signature node containing the subagent's number. When `gen()` is invoked, definitions for all subagents are output.

Because of the possibility of subagent extraction, the `SYagent` class must provide facilities for collecting a variant's variables and converting them to global scope if they are referenced in a subagent. This symbol-table-within-a-symbol-table is implemented by three methods:

- `prep(`*variant*`)`: sets up the symbol entry so that `prep()` and `gen()` will utilize symbols for the designated variant number.
- `addVar(`*name*`,`*global*`)`: invoked whenever a variable name is encountered in a channel input context during the `prep()` subphase of code generation.

- `ref(`*name*`,`*global*`)`: obtains the C++ name of the variable, which gets "uniquified" if the variable is used in subagents. `ref()` detects the error of using the value of a variable before setting it (`ref()` without a prior `addVar()`).

Two more methods provide start- and end-of-code-block generators for the variant specified by `prep(`*variant*`)`: `genAgentProc(ostream&)` and `genEndAgent(ostream&)`.

4.4.2. SYaction entry. One `SYaction` object is created for each atomic action or channel. Atomics are subject to a limited variant phenomenon in that, if they are subscripted, all subscripts actually used must be recorded so that `ActionRef` objects can be output by `gen()`. The `ref(`*subs*`)` method serves this purpose, by both recording the subscripts and also returning the C++ name of the corresponding `ActionRef`.

Aside from `ActionRefs`, `gen()` also outputs `Channel` and `Atomic` definitions. Each definition is preceded by a block of preprocessor code that tests whether the user has provided a compile-time definition of the symbol *action*`_p`. If so, the symbol's value is used as the name of an external `ActionProc` to be linked with the action. If not, the default value of zero suppresses external linkage.

4.4.3. SYdatum entry. One `SYdatum` object is created for each unique `Datum` or `DatumVar` name. The first occurrence fixes the number of subscripts, and subsequent occurrences are validated via the `Symbol::argsOK()` method. Invoking `gen()` outputs a `DATUMDEF` macro with the appropriate number of subscripts.

5. Code generation phase

When the code generation phase in the `main()` function of cspm.y is reached, the parse tree is complete and all symbols have been collected. The symbol tables are not, however, in their final forms, since the subsequent extraction of subagents may cause some variables to be globalized (names changed), and, of course, new subagent names to be added, as described in Section 4.4.1 above.

Code generation has two subphases: (1) Generate C++ code from the parse tree to a scratch file. (2) Generate all the symbol definitions to the output file. This is followed by copying the scratch file to the output file to complete the translation.

The parse tree at its top level is simply an `LOPNP`, having one `PNdefn` node for each statement in the CSP specification. Generation involves calling `gen()` on each `PNdefn` node in turn, and checking the return code in case an error has occurred. Thus, `PNdefn::gen()` can be regarded as "translation headquarters" at the agent definition level. Most errors print a diagnostic and abort the translation without attempting to carry on any further.

Generating an individual `PNdefn` node is two-step process, consisting of a `prep()` step followed by a `gen()` step. The purpose of the prep is threefold:

1. to extract subagents where required (`PNcompose`)
2. to note the occurrence and scope of input variables (`PNvar`)
3. to bind agent constants to agentproc signatures (`PNconst`)

In general, calls to `prep()` are simply relayed down the tree by complex nodes to leaf nodes. However, if a node type knows that no candidates can lie below it, it can limit descent by returning a good status.

Subagent extraction is needed whenever a complex agent expression is found where a simple operand is required. In such cases, the `prep()` method extracts the subtree as a new agent, and invokes `prep()` and `gen()` on it, so that it physically appears in the translated output separately from its parent. It then substitutes the name of the extracted subagent for the subtree. This process of extraction can recurse as deeply as necessary, resulting in a series of nested `prep()` calls and finally a `gen()` at the lowest level, and so on back up the tree.

Following generation of the syntax tree to the scratch file, the symbol definitions, now in final form, are generated to the output file. First, there are some stock header statements output for `#include` files, then the `gen()` methods are invoked on the three global symbol tables. It finally remains only to copy the scratch file to output, and append the default main program (by #including `main.h`), which will process any command-line arguments and then start execution of the compiled system at the agent named SYS.

6. Future work and conclusion

For the sake of rapidly creating a serviceable translator, certain areas were intentionally slighted. One is the common CSPm syntax for external choice, P [] Q, which is not allowed. In order for the runtime framework to decide between P and Q, their initial events must be identified. It was far easier to make the translator require this disclosure from the user, i.e.,

a->P' [] b->Q', than to derive the events from arbitrarily complex process definitions. That is to say, cspt treats [] like the CSP prefix choice operator | (which does not appear in CSPm). P [] Q is certainly more general, and support for it could be added.

A second area is optimization, which is currently quite minimal. As explained above, cspt is capable of distinguishing which process invocations can be bound at translation time from those that require runtime binding. This kind of optimization only requires keeping track of the available process definitions.

On the other hand, cspt does not analyze the specification globally. For example, given `P=chan!22...` and `Q=chan?x...`, plus the composition `P[|{|chan|}|]Q`, it is clear that these references to `chan` can be bound at translation time. But if P or Q are also referenced in other contexts—say `P[[chan<-foo]]`, meaning that "chan" is renamed to "foo" within this invocation of P—then it is no longer safe to blindly generate code for P that sends output on "chan". Since the context of a given process invocation can evolve dynamically as the specification is executed—perhaps with multiple, different contexts—all event and channel references are currently left for runtime binding. This means that specifications having simple, static process architectures now pay an execution overhead penalty which could, in principle, be relieved by more intelligent translation-time analysis.

Another untapped area for development is support for non-integer data types. CSPm can deal with sequences, sets, and tuples. This means handling its channel, datatype, and subtype statements, which are rather complex and include set comprehension.

Developers may be interested in adding support for currently untranslated CSPm operators, such as replicated operators. It is also possible to extend CSPm—which means inventing syntax—and, indeed, support for some operators of Timed CSP [13] will be introduced with the upcoming version 5 of CSP++.

In summary, the cspt translator has the potential to be useful to CSP tool builders, and is offered as open source. With guidance from this paper, it should be possible to make sense of the source code and carry out worthwhile modifications and enhancements.

References

[1] C. A. R. Hoare. *Communicating Sequential Processes*. Prentice Hall Series in Computer Science. Prentice Hall, Englewood Cliffs, NJ, 1985.

[2] William B. Gardner. Converging CSP specifications and C++ programming via selective formalism. *ACM Trans. on Embedded Computing Sys.*, 4(2):302–330, 2005.

[3] W.B. Gardner. Bridging CSP and C++ with selective formalism and executable specifications. In *Proc. First ACM and IEEE International Conference on Formal Methods and Models for Co-Design (MEMOCODE '03)*, pages 237–245, June 2003.

[4] W.B. Gardner. *CSP++: An Object-Oriented Application Framework for Software Synthesis from CSP Specifications*. PhD thesis, Department of Computer Science, University of Victoria, Canada, 2000. Available from: http://www.cis.uoguelph.ca/ wgardner/.

[5] William B. Gardner and Micaela Serra. CSP++: A framework for executable specifications. In M. Fayad, D. Schmidt, and R. Johnson, editors, *Implementing Application Frameworks: Object-Oriented Frameworks at Work*, chapter 9. John Wiley & Sons, 1999.

[6] W. B. Gardner. CSP++: How Faithful to CSPm? In *Proc. Communicating Process Architectures 2005 (WoTUG-27)*, Concurrent Systems Engineering, pages 129–146, Eindhoven, September 2005. IOS Press.

[7] Mantis H.M. Cheng. Communicating Sequential Processes: A synopsis. Technical report, Department of Computer Science, University of Victoria, April 1994.

[8] Stephen Doxsee. Reengineering CSP++ to conform with CSPm verification tools. Master's thesis, Department of Computing and Information Science, University of Guelph, Guelph, ON, Canada, August 2005.

[9] S. Doxsee and W.B. Gardner. Synthesis of C++ Software from Verifiable CSPm Specifications. In *12th IEEE International Conference and Workshops on the Engineering of Computer-Based Systems, 2005. ECBS '05*, pages 193–201, 2005.

[10] Bryan Scattergood. *The Semantics and Implementation of Machine-Readable CSP*. PhD thesis, University of Oxford, Oxford University Computing Laboratory, Programming Research Group, 1998.

[11] FDR2 user manual [online, cited 1/4/07]. Available from: http://www.fsel.com/documentation/fdr2/html/.

[12] A. W. Roscoe. *The Theory and Practive of Concurrency.* Prentice Hall PTR, Internet edition, 2005. Available from: http://web.comlab.ox.ac.uk/oucl/work/bill.roscoe/publications/68b.pdf.

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