COMMUNICATING SEQUENTIAL PROCESSES FOR SOFTWARE DESIGN MODELLING

Qiu Siqi¹, Koh Chuen Hoa²
¹ NUS High School of Mathematics and Science, 20 Clementi Ave 1, Singapore 129957
² DSO National Laboratories, 20 Science Park Drive, Singapore 118230

ABSTRACT

The current study investigates the possibility of using Communicating Sequential Processes (CSP) as a modelling language to provide more precise specifications to the implementation of a program in C language. In doing so, we write a program to convert a C source code into a CSP program such that the implementation can be directly compared with the initial specification.

INTRODUCTION

Software systems interact with us every day. As technology advances and we become more reliant on them, the complexity of software systems grow larger, yet so does the negative impacts software failure can bring us. The need of developing correct and reliable software is a big problem. As such, the development of system verification tools is a major research topic in software engineering[1]. Software verification can be dynamic or static. Here, we are interested in a subset of static verification known as formal verification.

Formal verification involves proving that a program satisfies a specification of its behaviour. In the developmental process, due to collaborative development, with a group of developers, it is difficult to ensure that all developers are able to follow and meet with the requirements of the designer. Typically, the designer provides a program specification which can state the gist of the product while guiding implementation in code. This is done through the development of a model prior to implementation. Unified Modelling Language (UML) is viewed as the standard language for software architecture development due to its ease of use and wide platform support[2]. However, the abstract syntax of graphical models such as UML results in imprecision and the formation of ambiguity which is unhelpful in facilitating understanding for parties involved[3]. This may result in implementation that does not meet the requirements. Other than that, it is difficult to take the model as a source of comparison to check conformance of final code to the specification.

We push for the use of CSP as a modelling language for specifying program design. CSP is a language for describing processes and patterns of interaction between them. It is supported by mathematical theory and a set of proofs[4]. CSP allows for formal specification of the program, which facilitates understanding as it is unambiguous and precise. This can better ensure the accuracy, reliability and maintainability of the project as a whole. Furthermore, there are various existing tools that allows for automatic verification and analysis of programs written in CSP such as PAT[5] and FDR[6]. Such tools help to ensure the consistency of the model without the need of rigorous manual proving.

We produce a script that is able to translate a C source code to a CSP program. This provides a straightforward way for the implementation of the program to be checked to comply with the rules and standards of the original design of the program as given in CSP. The overall
control flow of the implementation should match with that in the model and safety properties should be retained.

**COMMUNICATING SEQUENTIAL PROCESS**

CSP is an event based notation that describes the sequence of behaviours within a process and the communication between different processes. Processes interact with each other via message passing using channels and synchronize with each other via events[7].

**Relevant Syntax**

\[ P(x,y,z) = \text{Exp} \]  — This defines a process named \( P \) with parameters \( x, y, z \) which behaves as specified by \( \text{Exp} \). \( \text{Skip()} \) terminates the process.

\[ e\{x = 0; \ y=1;\} \rightarrow P(); \]  — This means that the event \( e \) is performed followed by process \( P \). Events are still valid without the name, though it will no longer be able synchronize with other processes. They are also valid without the statement block.

\[ \text{channel } g \ 0; \]  — This defines a channel \( g \) with buffer size 0 which means it sends or receives messages synchronously. Once a message is put into the channel, another process must take it out immediately.

\[ P() = g!x.y \rightarrow g\_return?z \rightarrow P(); \]  — \( g!x.y \) means to place the compound value of \( x \) and \( y \) into the channel buffer of \( g \), waiting if it is full. \( g\_return?z \) means to retrieve the top element on the buffer of \( g\_return \), waiting if it is empty. This represents a function call \( g(x,y) \) with return value \( z \). Function \( g \) has to be defined. It retrieves from the input channel and puts into the return channel instead. – \( \text{Func()} = g?x.y \rightarrow g\_return!z \rightarrow \text{Func();} \)

\[ P() = (([x<0] \{x = x+1;\} \rightarrow P())[*](\text{Skip()})); \]  — \( [*] \) is the choice operator. Here, it means that either addition or skip may execute. \( [x<0] \) is a guard condition such that the process after it only executes if the condition is satisfied. This example represents a loop with condition \( x<0 \).

\[ P() \parallel Q() \]  — This means that \( P \) and \( Q \) run concurrently in parallel.

**Concurrent systems**

CSP is a formal method that focuses on concurrent systems. Several subsystems mostly operate independently while collaborating by sending messages or having an event in common. This is similar to real world complex systems with independent subsystems that minimally communicate. A typical example is an abstract representation of a vending machine and its interaction with a person[8].

\[ VM() = \text{coin} \rightarrow (\text{tea\_button} \rightarrow \text{tea} \rightarrow \text{STOP}() \parallel [\text{coffee\_button} \rightarrow \text{coffee} \rightarrow \text{STOP}()]) \]

\[ \text{Person()} = \text{coin} \rightarrow \text{tea\_button} \rightarrow \text{STOP}() \]

\( VM \) models a vending machine that use \( \text{coin} \) event to detect when a person has inserted a coin into the machine. \( \text{Person} \) models a person that use the same \( \text{coin} \) event to represent insertion of coin into the machine.

\[ \text{System()} = VM() \parallel \text{Person()} \equiv \text{coin} \rightarrow \text{tea\_button} \rightarrow \text{tea} \rightarrow \text{STOP}() \]
This two processes can be put in parallel so that they can interact with each other. When two processes or more communicate via an event, that event will either occur simultaneously in both processes, or it won’t occur in either. Since the person only pushes the tea button, the composite process System is only able to dispense tea.

**Sequential representation**

While CSP is popularized by parallel programming, which is unsurprising given its focus on concurrency, its link to sequential C programs may not be so clear. That said, CSP is indeed capable of outlining a sequential program.

<table>
<thead>
<tr>
<th>CSP program</th>
<th>C program</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>var mx; var my;</code></td>
<td>`int g(int x) {</td>
</tr>
<tr>
<td><code>channel g 0; channel g_return 0; channel f 0;</code></td>
<td>if (x &gt; 0) { return x; }</td>
</tr>
<tr>
<td><code>channel f_return 0;</code></td>
<td>} else { return -x; }</td>
</tr>
<tr>
<td><code>channel main 0; channel main_return 0;</code></td>
<td>}</td>
</tr>
<tr>
<td>`G() = g?x -&gt; (</td>
<td>`int f(int x) {</td>
</tr>
<tr>
<td>(x &gt; 0) function_g_branch_1 -&gt; g_return!x</td>
<td>if x = f(5);</td>
</tr>
<tr>
<td>-&gt; G()) [*] (function_g_branch_2 -&gt; g_return!-x -&gt;</td>
<td>int my = g(3);</td>
</tr>
<tr>
<td>G())</td>
<td>return mx + my;</td>
</tr>
<tr>
<td><code>F() = f?x -&gt; g!x+1 -&gt; g_return?f_return_result_1</code></td>
<td>`Main() = f!5 -&gt;</td>
</tr>
<tr>
<td>f_return!f_return_result_1 -&gt; F();`</td>
<td>f_return?main_return_result_1 -&gt;</td>
</tr>
<tr>
<td>`Main() = f!5 -&gt;</td>
<td>{ mx = main_return_result_1; } -&gt;</td>
</tr>
<tr>
<td>f_return?main_return_result_1 -&gt;</td>
<td>g!3 -&gt; g_return?main_return_result_2</td>
</tr>
<tr>
<td>{ my = main_return_result_2; } -&gt;</td>
<td>{ main_return_result_2; } -&gt;</td>
</tr>
<tr>
<td>main_return!(mx + my) -&gt; Main();`</td>
<td>main_return!(mx + my) -&gt; Main();`</td>
</tr>
</tbody>
</table>

The example above shows how CSP is closely related to a C program. Although the processes are put in parallel, the processes are waiting for messages in the channels and therefore the whole system is sequential like the C program. CSP is able to precisely express the flow in the program and therefore can be a reliable means of guiding software implementation. While it may look too complicated to be a modelling language, it is to be noted that `G()` can be called `abs()` while `F()` can be called `add1_abs()` such that it is understandable what they perform and their implementation can be left out in the model. The model will contain more imaginary functions or invisible events, and only the key details.

**Process Analysis Toolkit (PAT)**

PAT is a self-contained framework to support composing, simulating and reasoning of concurrent systems. It is designed to incorporate advanced model checking techniques to
analyse event-based compositional system models\cite{5}. PAT provides refinement checking, simulation and dedicated algorithms to analyse specialized properties. With the inclusion of assertions, PAT’s verifier is able to automatically check a design for the wanted properties and provide a counterexample in the form of a trace if the property is not achieved.

```c
#assert System() deadlockfree;
```

A deadlock state is a state with no further move, but is not the successfully terminated state. This is usually a situation where the communicating processes are each waiting for the other to finish and therefore neither does. Systems that are not deadlock free may sometimes not be able to terminate successfully or run forever depending on the choices taken.

```c
#assert System() reaches goal;
```

This checks that the system is able to fulfil a certain functional criteria at any point. For example, in a river crossing puzzle solver, there has to exist a point where everyone is across.

**METHODOLOGY**

We wrote a program in C++ that reads a C source code line by line and processes each line independently. This is done with the help of standard C++ libraries. The program picks up important bits in the source code and translates lines that contain them into the corresponding output in CSP. Since we want to ensure that the implementation follows the overall flow of logic as in the model, we are interested in lines that shows a change in the control flow of the implemented program. This includes conditional statements, loops and function calls.

The program works on certain assumptions:

1. There are no switch-case statements.
2. There are no do-while loops.
3. Function prototypes are not used and function dependencies are defined above.
4. No 2 variables are of the same identifier regardless of their scope.
5. There is at most one level of branching within a function. Nested branching can be achieved by making calls to functions.

For each line in the source code, there are 4 main parts in the processing.

The first part is where the line is modified in preparation for keyword search. The line is read character by character to remove strings and comments in the source code which may affect the accuracy of keyword search. A line that only contains spaces after modification does not go on to the next part of processing. This is because a meaningful instance of previous line, which will be stored after keyword search, is required. Counting of curly braces ({, }) also happens here. Braces counting is used to identify different blocks in the program. A count of 1 means that the current line or the previous line has a function header and represents the start of a function. With the help of stacks containing the count upon the opening of conditional blocks and loops, we are able to identify the end of these blocks and loops where specific instructions are written to the CSP output.

The second part of processing is handling of special cases. There are 3 special cases: opening of a new function, end of the current function and end of a loop. In these cases, we assume that the same line does not contain any of the keywords. For function declaration, in the CSP output, we make input and return channels, declare the start of a new process and determine
the arguments, if any. The function name is retained for easy comparison with the initial model. For the end of a function, we stop writing to the process sequence by making it call itself and declare all variables and loops used in the function which are stored in memory prior to writing to the CSP file. For the end of a loop, we write (to memory) the increment if any, and the choice to jump out of the loop if a condition exist. Reference is also made to the outer loop, if any, or the function itself, to determine where to write the translation to. Lines which fall under these special cases do not go on to the next part.

The third part of processing is a search for function calls. Function calls are represented by function(args) – a name linked to an open bracket, whereby this name is not a reserved word. This function is assumed to be defined earlier in the same source code, but in the case it is not, it is treated as an imaginary function call in the CSP program with understandable meaning and is also assumed to have a return value. The search for function calls is completed using recursion. The initial string is returned when it contains no opening bracket. The recursive function finds the first instance of an opening bracket, finds the closing bracket that matches with it and calls itself firstly with the string enclosed in the brackets and next with the string after the closing bracket. The string right before the opening bracket is obtained and checked to not be space or a reserved word. If it is indeed a function call, the function call and return value will be written in the CSP output and the line in our program memory will be modified such that it contains the return value:

\[ x = \text{function1(function2(args))} > x = \text{function1(function2_return)} > x = \text{function1_return} \]

The last step in processing is keyword search. Keyword search is a straightforward instance of finding conditional statements – “if” and “else” and loops – “while” and “for”, extracting their conditions and writing it in CSP. Information that are to be used at a later point such as for loops’ increment are stored.

**EXAMPLE**

Given that there are only a few pieces of hardware, when there are many processes that require the usage of one hardware at the same time, for instance when you need to talk to different parties over the network using a single port, it is important that the hardware program is able to coordinate the access properly. It is also helpful to model what the software is supposed to do sequentially because the order of operation and the presence of certain operations are important so that it does not affect other processes using the same hardware, for instance the need to free a port after use.

We model a hardware that is shared by two different processes.
Below are some properties of the system:

1. Hardware can be idle or busy.
2. The hardware supports the functions CheckStatus, DoWork, IsDone and RetrieveAnswer.
3. To use the hardware, processes first need to CheckStatus.
4. If the hardware is idle, the process can ask it to DoWork, after which the hardware turns busy. If the hardware is busy, the process should go on to do other things.
5. IsDone and RetrieveAnswer can only be called by the requester, to check if the job is done and to complete the job so that the hardware falls back to idle respectively.

We implement a minimal version of \textit{Function} in C together with the methods supported by the hardware as in-place methods for the actual hardware. The implementation mainly shows instances of interaction between the process and the hardware.
In Figure 2, all *etc* calls are placeholders to represent other operations that may take place other than hardware calls.

In Figure 3, the timer function is not shown. *workDone* is toggled to true when a certain amount of time has passed.

Using our translator program, we extract a CSP program out of the C implementation of *Function*. We add the supported hardware functions into the file so that these functions will be reflected as existing in the extracted CSP output. Figure 4 below shows the portion of the output corresponding to *Function* in the model.
In Figure 4, all *etc* calls are not extracted because they lack communication with the process in terms of input and output.

We can now check if the implementation is accurate through the similarity in control flow (branches that can be taken). More specifically, in this case, we want to assure that the programmer did not accidentally omit or re-order the hardware calls. In Figure 4, we see that *IsDone* is assured before *RetrieveAnswer* is called and *CheckStatus* is called and ‘idle’ is returned before *DoWork* is called. This is as specified in the model shown in Figure 1.

**FUTURE WORK**

We can write a script that rearranges any C program into one that follows a rigid convention and fulfils our program’s assumptions. Also, we can modify the code to reduce the assumptions, especially to support function prototypes since it may not be the case where all functional dependencies can be defined at an earlier stage for all functions. Include statements may be considered such that all functions need not be in the same source code. We can test this idea on other programming languages apart from C to better justify the potential of CSP as a modelling language.

**CONCLUSION**

We provide a means for automatic generation of a CSP program from the C program and shown that this output should be similar to the initial design written as a CSP program, except that it may be more detailed and complete in content. With this, we are able to ensure that the implementation of the program conforms to its specifications.

While using CSP as a modelling language offers more precision, it comes with the price of increased reliance on the designer and it may not be immediately obvious how to describe the system in CSP. The integration of both graphical and formal notation may be useful in promoting specification comprehensibility as compared to having the formal notation alone[9]. As such, CSP can act as a useful enhancement rather than replacement to graphical modelling means.
REFERENCES


FURTHER READING
