Simulating CSP-Like Languages

Thomas Gibson-Robinson

Department of Computer Science, University of Oxford

August 25, 2013

Introduction

Building a model-checker for a process algebra requires a major investment of resources. Therefore, it would be highly desirable to be able to re-use existing model checkers where possible.

In *On the Expressiveness of CSP*, Roscoe proves that a large class of process algebras can be simulated in CSP. Unfortunately, Roscoe's simulation was not directly implementable in machine-CSP for a variety of reasons.

Contributions

We make three main contributions:

- Roscoe's simulation has been modified to allow it to be encoded in machine-CSP. The simulation efficiency has been drastically improved to allow realistic sized processes to be considered.
- The simulation has been modified to allow a large class of recursive processes to be simulated.
- A tool has been developed to allow the simulation to be automatically constructed given *natural* descriptions of the semantics of the process algebra and the script to be simulated.

CSP-Like Process Algebras

Consider the CSP operators (e.g. $\rightarrow$, $\Box$, $\sqcap$, $|||$, $\triangle$, Θ_A):

- An operator only performs an action as a result of one of its arguments performing an action.
- If we define ***on*** arguments as those that can perform a visible event, then precisely the **on** arguments are allowed to do τ's (correspondingly, an operator that cannot do a visible event is ***off***). For example:

$$\frac{P \xrightarrow{\tau} P'}{P \Box Q \xrightarrow{\tau} P' \Box Q}$$

- Each **on** argument appears only once in the resulting state. For example, there is no rule such as:

$$\frac{P \xrightarrow{a} P'}{Clone(P) \xrightarrow{a} P \,|||\, P'}$$

CSP-Likeness

An operator $Op(P_1, \ldots, P_N)$ is CSP-like iff its operational semantics can be written as a set of rules (ϕ, e, P) where:

- ϕ is a partial map that indicates which argument of Op has to perform which event;
- e is the event that Op performs;
- P is the resulting state of the process.

For example, the rules of $P \,\Box\, Q$ can be expressed as:

$$(\{1 \mapsto a\}, a, ID(\mathbf{1})) \qquad a \in \Sigma$$
$$(\{2 \mapsto a\}, a, ID(\mathbf{2})) \qquad a \in \Sigma$$

Whilst the rules of $P \,\Theta_A\, Q$ can be expressed as:

$$(\{1 \mapsto a\}, a, \mathbf{1} \,\Theta_A\, \mathbf{2}) \qquad a \in \Sigma \setminus A$$
$$(\{1 \mapsto a\}, a, ID(\mathbf{2})) \qquad a \in A$$

The Simulation

Let Op be a CSP-like operator and **OnProcs** / **OffProcs** be vectors of processes. Consider defining a CSP process $Operator(Op, \mathbf{OnProcs}, \mathbf{OffProcs})$ to simulate Op when applied to **OnProcs** and **OffProcs**.

In order to simulate Op correctly, $Operator$ must:

- Offer those events that Op initially offers;
- When an event a is performed, $Operator$ must turn **off** processes that are now discarded (e.g. $\Box$) and turn **on** processes that are newly created (e.g. $\rightarrow$ or Θ_A).

Roscoe's Simulation

Roscoe defined a CSP process $Operator(Op, \mathbf{OnProcs}, \mathbf{OffProcs})$ that returns a strongly bisimilar CSP process.

$$Operator(Op, \mathbf{OnProcs}, \mathbf{OffProcs}) \mathrel{\widehat{=}}$$
$$\Big(\|_{(n,P)\in\mathbf{OnProcs}} (Harness(P, n), AlphaProc(n))$$
$$\|_{\ldots} Regulator(Op, \mathbf{OffProcs}) \Big) [\![Rename]\!] \setminus \{tau\}.$$

Internally, the simulation uses events of the form (ϕ, x, B) where:

- ϕ is a partial map that indicates what event each **on** argument performs;
- x is the event that Op performs;
- B is the set of processes that are turned **off**.

Efficiency of the Simulation

Performing a naïve translation of the above into CSP_M produces a simulation that is so inefficient that it cannot be used for anything but the simplest of examples.

On inefficiency is that the set of all events of the form (ϕ, x, B) is exponential in both the number of **on** processes and the size of the alphabet.

However, note that $|||$ only requires events of the form:

$$(\{1 \mapsto a\}, a, \{\}) \qquad (\{2 \mapsto a\}, a, \{\})$$

Further, all CSP operators only require polynomially many events in terms of the size of the alphabet.

Improving Efficiency

The second problem is that the $Harness$ definition is rather complex:

$$Harness(P, n) \mathrel{\widehat{=}} \Big(\big(P[\![\ldots]\!] \; \Theta_{\ldots} \; STOP \big) \triangle \mathit{off} \rightarrow STOP \Big) [\![\ldots]\!]$$

This is a complex expression for FDR to compile, and (in practice) significantly slows it down.

However, many operators in CSP and other process algebras have arguments that they never turn **off**, meaning that the full harness does not need to be used for these arguments.

Recursion

Unfortunately, when the modified simulation is applied to recursive processes, the resulting simulation cannot be compiled by FDR. For example, consider the simulation of $P = a \rightarrow P$:

$$\begin{aligned} P &= Operator(Prefix.a, \langle\rangle, \langle P \rangle) \\ &= Reg(Prefix, \ldots)[\![R]\!] \\ &= ((\{\}, a, \{\}) \rightarrow (Harness(P, 0) \underset{\ldots}{\|} Reg(Identity, \ldots)))[\![R]\!] \\ &= a \rightarrow (Harness(P, 0) \underset{\ldots}{\|} Reg(Identity, \ldots))[\![R]\!] \\ &= a \rightarrow (P[\![HR]\!] \underset{\ldots}{\|} Reg(Identity, \ldots))[\![R]\!]. \end{aligned}$$

Thus, P_i, which is P unwrapped i times, is equivalent to:

$$\begin{aligned} P_0 &= P \\ P_{i+1} &= a \rightarrow ((P_i[\![HR]\!] \underset{..}{\|} Reg(Identity, ...))[\![R]\!]). \end{aligned}$$

Recursion

Thus, in general, the simulation fails because recursion causes an infinite number of processes to be put in parallel, meaning that FDR never finds the fix-point.

One solution to this would be to *throw-away* the collection of accumulated identity regulators whenever we do a recursion:

$$Q \mathrel{\widehat{=}} Operator(Prefix.a, \langle\rangle, \langle recurse \rightarrow STOP\rangle)$$
$$\Theta_{\{recurse\}}\, Q$$

This will require the following rule to be added to the identity operator:

$$\frac{P \xrightarrow{recurse} P'}{Identity(P) \xrightarrow{recurse} Identity(P')}$$

Thus:

$$
\begin{aligned}
Q \mathrel{\widehat{=}} & \mathit{Operator}(\mathit{Prefix}.a, \langle\rangle, \langle \mathit{recurse} \to \mathit{STOP} \rangle) \\
& \quad \Theta_{\{\mathit{recurse}\}}\, Q \\
= & \left[a \to ((\mathit{recurse} \to \mathit{STOP})[\![HR]\!] \underset{\ldots}{\|} \mathit{Reg}(\mathit{Identity}, \ldots))[\![R]\!] \right] \\
& \quad \Theta_{\{\mathit{recurse}\}}\, Q \\
= & \left[a \to \mathit{recurse} \to (\mathit{STOP}[\![HR]\!] \underset{\ldots}{\|} \mathit{Reg}(\mathit{Identity}, \ldots))[\![R]\!] \right] \\
& \quad \Theta_{\{\mathit{recurse}\}}\, Q \\
= & \; a \to \mathit{recurse} \to Q
\end{aligned}
$$

Hence, $Q \setminus \{\mathit{recurse}\} = \mu\, X \cdot a \to X$.

Generalising The Solution

In order to generalise the solution we need to consider the different ways in which operators treat their arguments.

Some arguments of operators may possibly perform an infinite number of visible events before the operator is discarded. For example, both arguments of $|||$ or the first argument of Θ_A. Such arguments are known as *infinitely recursive*.

Conversely, some arguments of operators may be guaranteed to only perform a finite number of visible events before the operator is discarded. For example, both arguments of $\Box$, the left argument of timeout, etc.

A (Sketched) Generalised Solution

The $callProc$ channel is defined and every call to a recursive process P is replaced by a $callProc.P$ event.

The process $WrapThread(Q)$ is defined using $\Theta_{callProc}$ to handle recursions, much like the above example using $recurse$.

Each *infinitely recursive* argument P of an operator Op is replaced by $WrapThread(P)$.

It can then be proven that the original simulation is equivalent in every CSP denotational model to the modified simulation.

Recursion (Example)

For example, the simulation of $P \mid\mid\mid Q$, given that $P = a \rightarrow P$ and $Q = b \rightarrow Q$ is:

$$
\begin{array}{l}
Operator(Interleave, \langle \\
\quad WrapThread(\\
\quad\quad Operator(Prefix.a, \langle\rangle, \langle callProc.P \rightarrow STOP \rangle)), \\
\quad WrapThread(\\
\quad\quad Operator(Prefix.b, \langle\rangle, \langle callProc.Q \rightarrow STOP \rangle)) \\
\rangle, \langle\rangle)
\end{array}
$$

By the above results, the resulting process is equivalent in every CSP denotational model to the original (non-compilable) simulation.

Tool Support

A tool, tyger, has been developed that automates the construction of the simulation, given the operational semantics of the process algebra to simulate and a script in that process algebra:

```
Operator Exception(P : InfRec, Q, A)
    Syntax Binary "[| $3 |>" 12 AssocNone
    Rule
        P =a=> P'
        ------------------------------ a <- diff(Sigma, A)
        P [| A |> Q =a=> P' [| A |> Q
    EndRule
    Rule
        P =a=> P'
        ------------------- a <- A
        P [| A |> Q =a=> Q
    EndRule
EndOperator
```

tyger

The tool takes as input two files, one consisting of the operational semantics of the language and the other containing a script to simulate.

- After parsing the first input file, tyger performs a sort of type-checking on the operational semantics and infers which arguments of the operators are **off** and **on**;
- The tool then outputs a file that contains the definition of *Operator*;
- The tool then type-checks the second file and uses the resulting information to perform the recursion refactorings on the file;
- The tool then constructs the simulation and pretty-prints in to a file.

Experiments: Simulation Overhead

The following table gives the time taken to check if the Dining Philosophers problem is deadlock free for various number of philosophers:

	Number of Philosophers							
Tool	3	4	5	6	7	8	9	10
Simulation	0.4	1.0	1.9	3.3	5.6	10.6	33.2	173.2
FDR	< 0.1	< 0.1	< 0.1	0.1	0.2	1.1	7.0	42.5

All experiments were carried out on a Linux Virtual Machine with 2GB of RAM and one 2.2Ghz Core i7 core. FDR 3.0-beta-3 was used in single-threaded mode.

Experiments: Simulating CCS

The following table gives the time taken to check if the Dining Philosophers problem is deadlock free by both the Concurrency Workbench (using version 7.1) and by a CSP simulation of CCS (running under FDR 3.0-beta-3 in single-threaded mode).

	Number of Philosophers						
Tool	3	4	5	6	7	8	9
CWB	< 0.1	< 0.1	0.3	2.6	38.5	*	*
Simulation	0.6	0.8	1.8	3.0	8.9	53.9	384.0

Summary

We have:

- Altered Roscoe's simulation to allow it to be be written in machine-CSP and to be efficiently manipulated by FDR;
- Developed a method of refactoring processes that allows a large class of recursive processes to be correctly simulated;
- Constructed a tool that can automatically construct the simulation given the operational semantics of a CSP-like process algebra and a script to simulate.