Communicating Process Architectures 2013
P.H. Welch et al. (Eds.)
Open Channel Publishing Ltd., 2013

Efficient Simulation of CSP-Like Languages

Thomas GIBSON-ROBINSON
Department of Computer Science, University of Oxford, UK
thomas.gibson-robinson@cs.ox.ac.uk

Abstract. In *"On the Expressiveness of CSP"*, Roscoe provides a construction that, given the operational semantics rules of a *CSP-like* language and a process in that language, constructs a strongly bisimilar CSP process. Unfortunately, the construction provided is difficult to use and the scripts produced cannot be compiled by the CSP model-checker, FDR. In this paper, we adapt Roscoe's simulation to make it produce a process that can be checked relatively efficiently by FDR. Further, we extend Roscoe's simulation in order to allow recursively defined processes to be simulated in FDR, which was not supported by the original simulation. We also describe the construction of a tool that can automatically construct the simulation, given the operational semantics of the language and a script to simulate, both in an easy-to-use format.

Keywords. CSP, CSP-like, FDR, model checking

Introduction

Developing a model checker for a process algebra requires a significant investment of resources, particularly if the model checker is to be efficient. Therefore, rather than developing new model checkers for each new process algebra, it would make sense to reuse existing model checkers, where possible.

In papers on the *expressiveness* of CSP [1,2], Roscoe develops a simulation that, given a *CSP-like* language, constructs a strongly-bisimilar process in the process algebra CSP [3,4,5]. Thus, as CSP has an efficient, industrial-strength, model checker FDR [6], it should now be possible to check models written in other CSP-like languages without having to develop a new model checker.

Whilst the name CSP-like might appear to imply that only a small class of languages can be simulated using [1,2], most of the CSP-like conditions are merely well-formedness conditions and thus many other process algebras, such as CCS [7], the π-calculus [8] and Lowe's Readyness Testing CSP extension [9], are either CSP-like, or can be easily converted into equivalent, but CSP-like formulations. Further, many denotational models for CSP, such as Lowe's Availability model [10], have CSP-like derived operational semantics that enable the model to be extracted using the standard CSP models.

The above would suggest that a tool that could automatically construct a simulation of processes written in a CSP-like language, given the operational semantics rules as input, would be of value. Unfortunately, Roscoe's construction in [1,2] was never intended to be used as an actual simulation. In particular, only non-recursive processes can be simulated within FDR as the simulated recursive processes are infinite state. The simulation also imposes a very high computational overhead, making it impractical for any non-trivial problem. Further, the simulation is extremely complex to construct and the user has to construct the simulation manually.

Contributions In this paper we explain how to alter Roscoe's simulation to allow it to be encoded in machine-CSP (henceforth CSP$_M$). We also give a number of optimisations to Roscoe's simulation to ensure that it can efficiently simulate processes. We then prove that the resulting optimised simulation is still strongly bisimilar to the original process. We also extend Roscoe's simulation to enable recursive processes to be simulated correctly by FDR. We then prove that, whilst this alteration is no longer strongly bisimilar, it results in a process that is equivalent in all CSP models to the original process.

We also describe the construction of a tool, tyger, that takes as input the operational semantics rules of the language (in a easy-to-use format) and a list of process definitions to compile (again, in a natural format). The tool automatically produces the CSP$_M$ simulation described above, and also automatically applies the necessary alternations to ensure that recursive processes can be successfully simulated. The output of this tool is thus a CSP file that can be passed to FDR, or any other CSP$_M$ tool.

This has a number of advantages. Not only does it mean that model checkers can be easily constructed for other process algebras, but it allows a language designer to experiment with the language whilst developing it. For example, suppose a user wants to create a process algebra to enable easy verification of a certain problem. Previously, to experiment with the language, an entirely new model checker would have to be constructed, making it difficult for the language designer to experiment with the language's capabilities during design. Now, a simulation of the language can be constructed trivially, allowing experiments to be run whilst the language is being developed and therefore providing feedback into the design.

Outline In Section 1 we give a brief overview of CSP, before considering what languages we are able to simulate within CSP by defining Roscoe's CSP-like condition. In Section 2 we explain Roscoe's original construction before giving a number of modifications that enable it to run efficiently. In Section 3 we discuss the problem with recursion, explain the refactorings required to permit recursive definitions and prove that the refactorings produce semantically equivalent processes. Further, we discuss the limitations of the recursion refactorings. In Section 4 we describe the construction of the tool. In Section 5 we detail the results of several experiments that were run in order to verify the effectiveness and performance of the simulation. Lastly, in Section 6 we draw conclusions and discuss future work.

1. Background

In this section we provide a brief overview of the fragment of CSP necessary to understand the remainder of the paper. We also define what it means for an operational semantics to be *CSP-like*, which represents the class of operational semantic languages that we can simulate.

1.1. A Brief Overview of CSP

CSP [3,4,5] is a *process algebra* in which programs or *processes* that communicate events from a set Σ with an environment may be described. We sometimes structure events by sending them along a *channel* . For example, $c.3$ denotes the value 3 being sent along the channel c. Further, given a channel c the set $\{|c|\} \subseteq \Sigma$ contains those events of the form $c.x$.

The simplest CSP process is the process $STOP$, that can perform no events and thus represents a deadlocked process. The process $a \rightarrow P$ offers the environment the event $a \in \Sigma$ and then when it is performed, behaves like P. The process $P \; \Box \; Q$ offers the environment the choice of the events offered by P and by Q. Alternatively, the process $P \sqcap Q$, nondeterministically chooses which of P or Q to behave like. Note that the environment cannot influence the choice, the process chooses *internally*. $P \rhd Q$ initially offers the choice of the events of P but can *timeout* and then behaves as Q.

Syntax	Meaning
$a \rightarrow P$	Performs the event a and then behaves like P.
$P \mathrel{\square} Q$	Offers the choice between P and Q.
$P \sqcap Q$	Non-deterministically picks one of P and Q to behave as.
$P \setminus A$	Runs P but hides events from A (which become τ).
$P \mathbin{\vert\vert\vert} Q$	Runs P and Q in parallel, enforcing no synchronisation.
$P \underset{A}{\Vert} Q$	Runs P and Q in parallel, enforcing synchronisation on events in A.
$P \; {}_A\Vert_B \; Q$	Runs P and Q in parallel, enforcing synchronisation on events in $A \cap B$.
$P[\![R]\!]$	Renames the events of P according to the relation R.
$P \;\Theta_A\; Q$	Runs P until an event in A occurs, at which point Q is run.
$P \mathbin{\triangle} Q$	Runs P, but at any point Q may interrupt P, perform an action and discard P.

Table 1. A summary of the syntax of CSP.

TThe process $P \; {}_A\Vert_B \; Q$ allows P and Q to perform only events from A and B respectively and forces P and Q to synchronise on events in $A \cap B$. The process $P \underset{A}{\Vert} Q$ runs P and Q in parallel and forces them to synchronise on events in A. The *interleaving* of two processes, denoted $P \mathbin{\vert\vert\vert} Q$, runs P and Q in parallel but enforces no synchronisation. The process $P \setminus A$ behaves as P but hides any events from A by transforming them into a special internal event, τ. This event does not synchronise with the environment and thus can always occur. The process $P[\![R]\!]$, behaves as P but renames the events according to the relation R. Hence, if P can perform a, then $P[\![R]\!]$ can perform each b such that $(a, b) \in R$. The process $P \mathbin{\triangle} Q$ initially behaves like P but allows Q to *interrupt* at any point and perform an event, at which point P is discarded and the process behaves like Q. The process $P \;\Theta_A\; Q$ initially behaves like P, but if P ever performs an event from A, P is discarded and $P \;\Theta_A\; Q$ behaves like Q.

Recursive processes can be defined either equationally, or using the notation $\mu\, X \cdot P$. In the latter, every occurrence of X within P represents a recursive call to $\mu\, X \cdot P$.

There are a number of ways of giving meaning to CSP processes. The simplest approach is to give an operational semantics. The operational semantics of a CSP process naturally creates a labelled transition system (LTS) where the edges are labelled by events from $\Sigma \cup \{\tau\}$ and the nodes are process states. The usual way of defining the operational semantics of CSP processes is by presenting *Structured Operational Semantics* (SOS) style rules. For example, the operational semantics of $P \;\Theta_A\; Q$ can be defined by the following inductive rules:

$$\frac{P \xrightarrow{a} P'}{P \;\Theta_A\; Q \xrightarrow{a} Q}\; a \in A \qquad \frac{P \xrightarrow{b} P'}{P \;\Theta_A\; Q \xrightarrow{b} P' \;\Theta_A\; Q}\; b \notin A \qquad \frac{P \xrightarrow{\tau} P'}{P \;\Theta_A\; Q \xrightarrow{\tau} P' \;\Theta_A\; Q}$$

The interesting rule is the first, which specifies that if P performs an event $a \in A$, then $P \;\Theta_A\; Q$ can perform the event a and then behave like Q (i.e. the exception has been *thrown*). The last rule is known as a *tau-promotion* rule as it promotes any τ performed by a component (in this case P) into a τ performed by the operator. The justification for this rule is that τ is an unobservable event, and therefore the operator cannot prevent P from performing the event.

CSP also has a number of *congruent* denotational semantics, where congruence means that the denotational semantics may either be extracted from the operational semantics, or calculated directly given the process. For example, the *traces* model represents each process by the prefix-closed set of finite sequences of visible events that it can perform (i.e. a process is represented by a subset of Σ^*).

1.2. CSP-Like Languages

In [1] Roscoe proves that if an operator Op is *CSP-like*, then it is possible to construct a CSP process Op' such that the LTSs representing Op and Op' are strongly bisimilar. In this section we define what it means for an operator to be CSP-like.

Conventionally, operational semantics are represented by SOS rules, such as those given above for the CSP exception operator. However, given that the syntax of SOS rules is very general, defining what CSP-like means in terms of the rules would be rather difficult. For example, the following SOS rule is not a legal CSP-like rule (since no CSP operator allows an event to occur as the result of the absence of another event), but is a valid SOS rule:

$$\frac{\neg(P \xrightarrow{a} P')}{Op(P) \xrightarrow{a} Op(P')}.$$

Therefore, in [1], Roscoe instead uses a different presentation of operational semantics rules in which only CSP-like operators are definable. We now give a brief overview of this alternative presentation: more information can be found in [1,5].

Definition 1.1 (From [1,5])**.** The *combinator* operational semantics of an operator $F(\mathbf{P})$, where $\mathbf{P}$ is a vector of processes, is defined by:

- Each process of the vector $\mathbf{P}$ is defined as either ***on*** or ***off***. **on** arguments are indexed by positive numbers, whilst negative indices refer to **off** arguments. Intuitively, an **on** argument of a process can perform visible events, whilst an **off** argument cannot. For example, both arguments of $\Box$ are **on**, the first argument of Θ_A is **on**, whilst the second argument of Θ_A is **off**.
- A set of *combinator rules* that specify the operational semantics of the operator, except tau promotion rules which are implicitly defined for precisely the **on** arguments. A single combinator rule is of the form (ϕ, x, T) where:
  - $*$ ϕ is a partial function from the **on** indices to Σ. If $\phi(x) = y$ then the x^{th} **on** process performs the event y. If $x \notin dom(\phi)$, then x does not perform an event.
  - $*$ $x \in \{\tau\} \cup \Sigma$ is the resulting event that the operator performs;
  - $*$ T is a piece of syntax that represents the state of the resulting process. T is generated from the following grammar:

    $$T ::= -\mathbf{N} \mid \mathbf{N} \mid Op(T, \ldots, T)$$

    where: the first clause represents an **off** argument, the second clause a currently **on** argument, and the third clause represents the application of an operator to a number of process arguments. Further, there are a number of well-formedness conditions on T:

    1. **on** processes cannot be cloned, i.e. each **on** index **i** can appear at most once in T (there is no CSP operator that can clone an **on** argument);
    2. **on** processes cannot be suspended, i.e. if T contains $Op'(P_1, \ldots, P_N)$ where $P_i = \mathbf{j}$ (i.e. P_i is the j^{th} currently **on** process of Op), then i must be an **on** argument of Op' (again, no CSP operator can suspend and then later reactivate an **on** argument).

    For example, if $T = \mathbf{1}$, then this represents the rule that discards the current operator and discards all process arguments except for $\mathbf{1}$. Alternatively, if $T = \mathbf{1} \Box -\mathbf{1}$, then the resulting state is the external choice of the 1^{st} argument of the current operator and the newly started process $-\mathbf{1}$.

For example, the external choice operator has two arguments which are both **on** and the combinator rules consist of the set of all rules of the form $((a, \cdot), a, \mathbf{1})$ and $((\cdot, a), a, \mathbf{2})$, where a ranges over Σ. The exception operator has two arguments, the first of which is **on** whilst the second is **off**. The set of combinator rules for Θ_A consists of the set of all rules of the form $((a, \cdot), a, \mathbf{1}\ \Theta_A\ \mathbf{-1})$ where $a \in \Sigma \setminus A$, and rules of the form $((a, \cdot), a, \mathbf{-1})$ where $a \in A$.

A *CSP-like* operator is then defined as any operator whose operational semantics can be expressed by a combinator operational semantics. A CSP-like language is defined as a language in which every operator is CSP-like. Whilst this might appear to be restrictive, it turns out that many process algebras, including CCS[1] [7] and the π-calculus [8], are CSP-like. Further, many denotational models for CSP, such as Lowe's Availability Model [10], have a derived operational-semantics that is CSP-like, meaning that we are able to simulate these denotational models operationally.

2. Simulating CSP-Like Languages

In this section we define Roscoe's simulation and then define an optimised CSP$_M$ simulation, before proving the equivalence of the two. We start in Section 2.1 by describing Roscoe's simulation, including the internal format of the operational semantics and the CSP process $Operator$ that simulates a particular operator. In Section 2.2 we then explain the modifications that we made in order to construct a CSP$_M$ simulation. Further, we detail the changes that we made to enable the simulation to run efficiently and prove that the optimisation still result in a strong bisimulation.

Throughout this paper we will need to enlarge the set of events Σ that the environment offers. Thus, for the remainder of this paper we will use Σ to denote the set of events that the user uses in their script; $\Sigma_0 \supseteq \Sigma$ to include all semantic events (such as $\checkmark$ to indicate termination in the case of CSP); $\Sigma_1 \supseteq \Sigma_0$ to include all internal events used by the simulation. We also use a special event $tau \in \Sigma_0$ that will simulate a semantic τ.

2.1. Roscoe's Simulation

In order to simulate arbitrary CSP-like operators in [1,2], Roscoe defines a data structure in which the operational semantic rules of an operator are encoded. In this section, we start by defining this encoding before explaining the simulation.

Roscoe's encoding is an expansion of the combinator semantics, as presented in Definition 1.1, in order to make the combinators more easy to work with. As with the combinator semantics, operators such as $\cdot \setminus A$ are considered as a *family* of operators, one for each $A \subseteq \Sigma$. Thus, the only arguments of an operator are **on** and **off** processes. Given an operator α we define $n(\alpha)$ as the number of **on** arguments and $I(\alpha)$ as the number of **off** arguments. Further, we index the **on** arguments by $\{1 \ldots\}$ and **off** arguments by $\{\ldots - 1\}$ (i.e. a positive index i refers to the i^{th} **on** process of α, whilst a negative index $-i$ refers to the i^{th} **off** process of α). The operational semantics of an operator α is given by a set of tuples $(\phi, x, \beta, f, \psi, \chi, B)$ where:

- ϕ: specifies what event each **on** process must perform in order for this action to be performed. Formally, it is a partial map from $\{1 \ldots n(\alpha)\}$ to Σ_0, where $\phi(x) = b$ means the x^{th} **on** process performs b. This corresponds precisely to ϕ in the combinator rules (cf. Definition 1.1).
- x: is the resulting event from Σ_0 that α performs.
- β: is the resulting operator that this process evolves into.

[1] CCS is not quite CSP-like as τ resolves choice. However, Roscoe shows how to simulate it in [1].

- f: denotes which **off** process of α is used for each newly turned **on** process. Formally, it is a map from $\{1 \dots k\}$ (where k is equal to the number of processes turned **on**) to $\{-I(\alpha) \dots -1\}$ where $f(x) = y$ indicates the x^{th} newly turned **on** process is a copy of the y^{th} **off** process.
- ψ: specifies what the process arguments of β are, which can either be currently **on** processes of α, or newly turned **on** processes (according to f). Formally, it is a map from $\{1 \dots n(\beta)\}$ to $\{1 \dots n(\alpha) + k\}$ where $\psi(y) = x$ indicates that the y^{th} argument of β is either, if $x \leq n(\alpha)$ then the x^{th} **on** argument of α, or otherwise the $x - n(\alpha)^{th}$ newly turned **on** argument (i.e. the $f(x - n(\alpha))$ **off** process of α).
- χ: specifies what the **off** arguments of β are in terms of those of α. Formally, χ is a map from $\{-I(\beta) \dots -1\}$ to $\{-I(\alpha) \dots -1\}$ where $\chi(y) = x$ means that the y^{th} **off** argument of β is the x^{th} **off** argument of α.
- B: is the set of **on** processes that are discarded (this was not present in Roscoe's model but it is added for clarity).

In the following, we assume that there exists a function $Rules$ that gives the rules for each operator in the language that is being modelled. Thus, $Rules$ is a function from the set of all operators to the powerset of tuples $(\phi, x, \beta, f, \psi, \chi, B)$ of the above form. For example, assuming that the $a \rightarrow \cdot$ operator is denoted as $\mathit{Prefix}.a$, and that $\cdot \underset{A}{\|} \cdot$ is denoted as $Parallel.A$, a partial specification of the $Rules$ function for CSP can be encoded as follows (where the identity operator is denoted by $\mathit{Identity}$):

$$\begin{aligned}
Rules(\mathit{Identity}) &\mathrel{\widehat{=}} \{(\{(1,a)\}, a, \mathit{Identity}, \{\}, \{(1,1)\}, \{\}, \{\}) \mid a \in \Sigma_0\} \\
Rules(\mathit{Prefix}.a) &\mathrel{\widehat{=}} \{(\{\}, a, \mathit{Identity}, \{(1,-1)\}, \{(1,1)\}, \{\}, \{\})\} \\
Rules(\mathit{Parallel}.A) &\mathrel{\widehat{=}} \{(\{(1,a),(2,a)\}, a, \mathit{Parallel}.A, \{\}, \{(1,1),(2,2)\}, \{\}, \{\}) \mid a \in A\} \\
&\quad \cup \{(\{(1,a)\}, a, \mathit{Parallel}.A, \{\}, \{(1,1),(2,2)\}, \{\}, \{\}) \mid a \notin A\} \\
&\quad \cup \{(\{(2,a)\}, a, \mathit{Parallel}.A, \{\}, \{(1,1),(2,2)\}, \{\}, \{\}) \mid a \notin A\}
\end{aligned}$$

For the rest of this paper we assume that a function $Rules$ has been defined for the language that we are attempting to model.

Operator Simulation We now define the process $Operator(\alpha, \dots)$ from [1,2] (with some minor alterations) that simulates the CSP-like operator α. The general form that the process will take is:

$$\begin{aligned}
&Operator(\alpha, \mathit{onProcs}, \mathit{offProcs}) \mathrel{\widehat{=}} \\
&\quad \Bigg(\|_{(n,P) \in \mathit{onProcs}} (Harness(P,n), AlphaProc(n)) \\
&\qquad \underset{\bigcup_{n \in \{1 \dots |\mathit{onProcs}|\}} AlphaProc(n)}{\|} \\
&\qquad Reg\left(\alpha, |\mathit{onProcs}|, id_{\{1 \dots |\mathit{onProcs}|\}}, id_{\{-|\mathit{offProcs}| .. -1\}}\right) \Bigg) [\![Rename]\!] \setminus \{tau\}.
\end{aligned}$$

In the above, each **on** process is run in a harness that will both ensure that it is turned **off** when necessary and that renames its events to allow the simulation to function correctly. The regulator process is responsible for ensuring that only events according to the current operator are allowed to occur.

Observe that the left hand side of the parallel in the above definition does not depend on the current operator, only the **on** processes. Therefore, only Reg differs between, e.g. $Operator(ExternalChoice, \langle P, Q\rangle, \langle\rangle)$ and $Operator(Parallel.\Sigma, \langle P, Q\rangle, \langle\rangle)$. However, these have very different synchronisation patterns; in the former P and Q do not synchronise whereas in the latter, P and Q synchronise on Σ. Therefore, it follows that the regulator needs to allow all possible synchronisation patterns to occur.

In order to ensure that we can simulate the two different synchronisation patterns, we need to rename the events of each **on** process to allow the processes to synchronise together in any possible way. The regulator could then be defined to allow only those synchronisations that correspond to the current operator. Thus, the general event form that will be used in the operator simulation will be a tuple (ϕ, x, B) where:

- ϕ is a partial map from **on** process to Σ_0 (i.e. if $\phi(x) = y$ the x^{th} process performs the event y);
- x is the resulting event;
- B is the set of **on** processes that are discarded.

As we have fixed the event format, we are now able to define $Rename$ as the function that renames each (ϕ, x, B) to x. Further, $AlphaProc(n)$ can be defined to be the set of all (ϕ, x, B) such that n is affected; i.e. $n \in dom(\phi)$ or $n \in B$.

We now consider the definition of $Harness(P, n)$. The harness needs to run the process P as if it were the n^{th} **on** process, renaming every event that it performs to one of the above form. It also needs to turn this process **off** when an event of the form (ϕ, x, B) where $n \in B$ is performed. Hence, we define $Harness(P, n)$ as follows:

$$Harness(P, n) \mathrel{\widehat{=}} \Big(\big(P[\![Prime]\!] \ \Theta_{\{x' \mid x \in \Sigma_0\}} \ STOP\big) \triangle \mathit{off} \rightarrow STOP\Big)[\![\mathit{HRename}]\!]$$

where *off* is a fresh event and $Prime$ maps every event $x \in \Sigma_0$ to both x and x'. ***HRename*** renames as follows:

- Each event $a \in \Sigma_0$ to every (ϕ, x, B) such that $\phi(n) = a$ and $n \notin B$ (i.e. where process n performs $\phi(n)$ and is not discarded);
- Each event $a' \in \{x' \mid x \in \Sigma_0\}$ to every (ϕ, x, B) such that $\phi(n) = a$ and $n \in B$ (i.e. process n performs a, but is discarded);
- off to every (ϕ, x, B) such that $n \notin dom(\phi)$ and $n \in B$ (i.e. process n is discarded by an event in which it does not partake).

We now define the regulator, which has three principle responsibilities: it must ensure that only events allowed by the current operator can be performed by the processes; it must track which process on the left hand side corresponds to which process of the current operator; and it must turn **on** processes according to the executed rule. Thus, the regulator takes the following parameters:

- λ: the current operator;
- m: the number of processes ever turned **on**;
- Ψ: a map from $\{1 \ldots n(\lambda)\}$ to $\{1 \ldots m\}$; in particular if $\Psi(x) = y$ it means that the the x^{th} **on** argument of λ is the y^{th} **on** argument;
- χ: a map from $\{1 \ldots I(\lambda)\}$ to $\{1 \ldots I(\alpha)\}$ where α is the starting operator (i.e. the operator for which $Operator(\alpha, \ldots)$ was called); in particular if $\chi(x) = y$ it means that the x^{th} **off** argument of λ is the y^{th} **off** argument of α.

We can then define the regulator, $Reg(\lambda, m, \Psi, \chi)$, as follows:

$$
\begin{array}{l}
Reg(\lambda, m, \Psi, \chi) \mathrel{\widehat{=}} \\
\qquad \Box_{(\phi,x,\beta,f,\psi,\chi',B) \in Rules(\lambda)} \, (\phi \circ \Psi^{-1}, x, \{\Psi(x) | x \in B\}) \rightarrow \\
\qquad\qquad \left(\begin{array}{c}
\|_{n \in \{m \ldots m+|f|\}} \, (Harness(\mathit{\mathbf{offProcs}}(\chi(f(n-m)))), AlphaProc(n)) \\
{}_{\bigcup_{n \in \{1 \ldots m+|f|\}} AlphaProc(n)}\|_{\bigcup_{n \in \{m+|f| \ldots\}} AlphaProc(n)} \\
Reg(\beta, m + |f|, (\Psi \cup id_{m+1,\ldots,m+|f|}) \circ \psi, \chi \circ \chi')
\end{array} \right)
\end{array}
$$

Note that the above regulator allows only those events that correspond to the current operator and, whenever such an event occurs, evolves into the state dictated by the rule that was chosen (with the state being updated according to the definitions of f etc.).

Roscoe then proves that the above simulation results in a CSP process that is strongly bisimilar to the original operator.

2.2. A CSP$_M$ Simulation

In this section we outline the alterations to Roscoe's simulation that are required in order to allow the simulation to be successfully modelled in CSP$_M$. The first major issue that prevents the above solution being compiled into CSP$_M$ is that the alphabets, such as $AlphaProc(n)$, are infinite because it is possible that an infinite number of processes can be turned **on** by an operator. Thus, since the **on** processes have to be able to be synchronised in any possible way, $AlphaProc(n)$ has to represent an infinite number of possible synchronisations patterns. In order to prevent this we disallow operators that could turn **on** an infinite number of arguments[2]. Hence, we can now statically bound the maximum number of processes turned **on** to some number N. Therefore, the set of all the (ϕ, x, B) events is given by:

$$\{(\phi, x, B) \mid \phi \in \{1 \ldots N\} \rightarrow \Sigma_0, x \in \Sigma_0, B \subseteq \{1 \ldots N\}\}.$$

We create a new CSP$_M$ channel, $renamings$ that carries events from the above set. Using this, the remainder of the translation to CSP$_M$ is largely mechanical and is therefore elided.

Efficiency Unfortunately, the naïve translation of the above into CSP$_M$ results in scripts that can only support at most a handful of events. In order to make the simulation run efficiently a number of optimisations had to be made. Some of the simpler optimisations included using sequences rather than sets to represent partial functions and using integers to identify events of the form (ϕ, x, B), rather than tuples. However, in order to make the simulation really run efficiently, a number of more interesting optimisations were made, as we now describe.

One problem with $Operator$ is that the $renamings$ channel has $O(2^{N+|\Sigma_0|} \cdot |\Sigma_0| \cdot 2^N)$ members, meaning that computing the $Harness$ renaming or the set $AlphaProc(n)$ will be slow. However, most operators will never use the majority of the events. For example, CSP's parallel operator (over a set A), requires only events of the form $(\{(0, a), (1, a)\}, a, \{\})$, $(\{(0, b)\}, b, \{\})$, $(\{(1, b)\}, b, \{\})$, where $a \in A$, $b \notin A$. In fact, all standard CSP operators only use $O(|\Sigma_0|^2)$ events. Thus, when computing the renamings and the process alphabets we restrict ourself to the subset of $renamings$ that could be used either by the current operator, or any resulting operator. This requires us to compute the set of all (ϕ, x, B) such that (ϕ, x, B) is permitted either by the current regulator, or any regular we can evolve to.

To calculate the above set, we firstly calculate the set of regulator configurations that the regulator can evolve into. In particular, we define the function $configs(\lambda, m, \Psi, \chi)$, where the arguments are as per Reg, as the least fixed point of the following function h:

[2] Clearly no practical simulation could support such operators, so this is not a restriction in practice.

$$\begin{aligned} h(X) \mathrel{\widehat{=}} X &\cup \{(\lambda, m, \Psi, \chi)\} \\ &\cup \{(\beta, m + |f|, (\Psi \cup id_{m+1,\ldots,m+|f|}) \circ \psi, \chi \circ \chi') \\ &\qquad \mid (\phi, x, \beta, f, \psi, \chi', B) \in \mathit{Rules}(\lambda), (\lambda, m, \Psi, \chi) \in X\}. \end{aligned}$$

Thus, it follows that $configs(\lambda, m, \Psi, \chi)$ consists of the set of all states that $Reg(\lambda, m, \Psi, \chi)$ can evolve into.

Using the above, the set of all events of the form (ϕ, x, B) that $Operator(\alpha, \mathit{onProcs}, \mathit{offProcs})$ requires, denoted $closure(\alpha, \mathit{onProcs}, \mathit{offProcs})$, can be defined as:

$$\begin{aligned} &\{(\phi \circ \Psi^{-1}, x, \{\Psi(x) | x \in B\}) \mid (\phi, x, \beta, f, \psi, \chi', B) \in \mathit{Rules}(\lambda), \\ &\qquad (\lambda, m, \Psi, \chi) \in configs(\alpha, |\mathit{onProcs}|, id_{\{1\ldots|\mathit{onProcs}|\}}, id_{\{-|\mathit{offProcs}|..-1\}})\}. \end{aligned}$$

The correctness of the above is shown in the following lemmas.

Lemma 2.1. Suppose $Operator(\alpha, \mathit{onProcs}, \mathit{offProcs})$ evolves into state in which Reg is in state $Reg(\lambda, m, \Psi, \chi)$. Then, $(\lambda, m, \Psi, \chi) \in configs(\alpha, |\mathit{onProcs}|, id_{\{1\ldots|\mathit{onProcs}|\}}, id_{\{-|\mathit{offProcs}|..-1\}})$.

Proof (Sketch). Observe that in the definition of $Reg(\lambda, m, \Psi, \chi)$, Reg evolves into states precisely the same as those on the left hand side of h in the definition of $configs$. □

Lemma 2.2. If $Operator(\alpha, \mathit{onProcs}, \mathit{offProcs})$ internally uses the event (ϕ, x, B) then $(\phi, x, B) \in closure(\alpha, \mathit{onProcs}, \mathit{offProcs})$.

Proof (Sketch). This follows from the observation that, internally, Reg participates in every event that $Operator$ performs, and thus the set of events that $Operator$ uses is bounded by the set of events that Reg allows, which, given Lemma 2.1, $closure$ extracts by definition. □

One other issue with the definition of $Operator$ is that it contains a large number of nested CSP operators, meaning that FDR has to deal with a large, complex, stack of operators. However, observe that if a particular component n can never be turned **off** (as is the case with CSP's parallel operator) then $Harness(P, n)$ is equivalent to $P[\![\mathit{Rename}]\!]$. Hence, if we can compute the set of arguments that can be turned **off**, it follows that we can easily decide when to compute the full harness. In order to compute the set of arguments that can be turned **off**, we define the function $turned\_off(\alpha, \mathit{onProcs}, \mathit{offProcs})$ as:

$$\begin{aligned} &\{\Psi(b) \mid (\lambda, m, \Psi, \chi) \in configs(\alpha, |\mathit{onProcs}|, id_{\{1\ldots|\mathit{onProcs}|\}}, id_{\{-|\mathit{offProcs}|..-1\}}), \\ &\qquad (\phi, x, \beta, f, \psi, \chi', B) \in \mathit{Rules}(\lambda), b \in B\}. \end{aligned}$$

The correctness of this optimisation can be proven as follows, noting that $Harness(P, n) = P[\![\mathit{Rename}]\!]$, if n can never be turned **off**. We elide the proof since it follows immediately from the definition of Reg and Lemma 2.1.

Lemma 2.3. If $Operator(\alpha, \mathit{onProcs}, \mathit{offProcs})$ evolves into a state in which $Harness(P, n)$ has turned P **off**, then $n \in turned\_off(\alpha, \mathit{onProcs}, \mathit{offProcs})$.

Having performed the optimisations we are now able to compile reasonably sized systems with little difficulty. Note that the above optimisations do not alter Roscoe's original result since, as proven above, the optimisations do not affect the semantics of the $Operator$ process.

3. Recursion

As discussed in the introduction, FDR cannot successfully compile simulations of recursive processes. For example, consider simulating CSP within CSP (which we do for ease of exposition only). According to the simulation given in Section 2, the process $P = a \rightarrow P$ would be simulated by the process $P = \mathit{Operator}(\mathit{Prefix}.a, \langle\rangle, \langle P\rangle)$. We calculate the definition of this as FDR would below, writing HR for the renaming done by the $Harness$ and R for the renaming done by $Operator$.

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

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

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

Whilst it is certainly true that P_i is equivalent to any of the other P_i, FDR (unsurprisingly) does not detect this and instead loops, looking for a transition back to P. Therefore, this results in an unbounded number of copies of $\mathit{Reg}(\mathit{Identity}, \ldots)$ being put in parallel.

In this section, we start in Section 3.1 by outlining a simple solution to the recursion issue outline above. In Section 3.2 we then generalise this to allow a large class of recursive definitions to be compiled. We then formalise the transformation and prove it correct in Section 3.3. In Section 3.4 we expand the transformation to permit additional recursive processes to be compiled. Lastly, in Section 3.5 we discuss the potential limitations of the recursion refactorings presented in this section.

3.1. A Simple Solution

One solution to the problem above would be to periodically *throw away* the collection of identity operator regulators that have been spawned. For example, consider the following simulation of $P = a \rightarrow P$:

$$\begin{aligned}
PSim &\mathrel{\widehat{=}} \mathit{Loop}(\mathit{Operator}(\mathit{Prefix}.a, \langle\rangle, \langle \mathit{callPSim} \rightarrow STOP\rangle))\\
\mathit{Loop}(Q) &\mathrel{\widehat{=}} Q \;\Theta_{\{\mathit{callPSim}\}}\; PSim.
\end{aligned}$$

When compiling $PSim$, according to the definition of Loop, whenever $\mathit{Operator}(\mathit{Prefix}.a \ldots)$ performs a $\mathit{callPSim}$ event the left argument of $\Theta_{\{\mathit{callPSim}\}}$ would be terminated and $PSim$ would be started again. In particular, if we calculate the definition of $PSim$ as FDR would, assuming that the identity operator allows $\mathit{callPSim}$ (which we discuss further below), we obtain the following:

$$\begin{aligned}
PSim &= \mathit{Loop}(\mathit{Operator}(\mathit{Prefix}.a, \langle\rangle, \langle \mathit{callPSim} \rightarrow STOP\rangle))\\
&= \mathit{Operator}(\mathit{Prefix}.a, \langle\rangle, \langle \mathit{callPSim} \rightarrow STOP\rangle) \;\Theta_{\{\mathit{callPSim}\}}\; PSim
\end{aligned}$$

$$
\begin{aligned}
&= (a \to ((callPSim \to STOP)[\![HR]\!] \underset{\dots}{\|} Reg(Identity, \dots))[\![R]\!] \setminus \{tau\}) \\
&\qquad \Theta_{\{callPSim\}}\ PSim \\
&= a \to (((callPSim \to STOP)[\![HR]\!] \underset{\dots}{\|} Reg(Identity, \dots))[\![R]\!] \setminus \{tau\} \\
&\qquad\qquad \Theta_{\{callPSim\}}\ PSim) \\
&= a \to (callPSim \to ((STOP[\![HR]\!] \underset{\dots}{\|} Reg(Identity, \dots))[\![R]\!] \setminus \{tau\}) \\
&\qquad\qquad \Theta_{\{callPSim\}}\ PSim) \\
&= a \to callPSim \to PSim.
\end{aligned}
$$

Hence, it follows that $PSim \setminus \{callPSim\}$ simulates $PSim$ in a way that FDR can successfully compile.

Note that $PSim \setminus \{callPSim\}$ is no longer strongly bisimilar to P as there is a τ transition whenever $PSim$ recurses. However, as all CSP denotational models equate $\tau \to P$ and P, this is not of any consequence providing the processes are used in refinement checks (which is the intention, after all).

3.2. Generalising the Solution

In order to generalise the solution of Section 3.1 to arbitrary processes and operators we give a number of refactorings that must be performed on the simulated script. The first refactoring will be as above; thus, we declare a new channel $callProc$ of type $Proc$ (for ease we assume process names may be sent over channels). We then replace every call inside a recursive process to another recursive process P with a $callProc.P$ event. Using this we can now define a process $WrapThread$ that is analogous to $Loop$ above. Note that much of the complexity in the following comes from the fact that the exception operator does not pass which event from the exception set was performed to its right argument. We work around this by putting the exception operator in parallel with a regulator that remembers which $callProc$ event occurred.

$$
\begin{aligned}
&WrapThread(P) = \\
&\quad ((P\ \Theta_{\{\!|callProc|\!\}}\ startProc?Q \to WrapThread(Q)) \\
&\qquad \underset{\{\!|callProc, startProc|\!\}}{\|} \quad \mu X \cdot callProc?p \to startProc!p \to X) \setminus \{\!|startProc, callProc|\!\}.
\end{aligned}
$$

Note that no $callProc$ events can propagate out of a process of the form $WrapThread(P)$.

As $\{\!|callProc|\!\} \not\subseteq \Sigma_0$ it follows that we will have to add the following rule to the operational semantics of the $Identity$ operator in order to allow the $callProc$ events can propagate, as discussed above.

$$
\frac{P \xrightarrow{callProc.p} P'}{Identity(P) \xrightarrow{callProc.p} Identity(P')}\ p \in Proc
$$

We might ask if similar propagation rules should be added to other operators for **on** arguments. However, it is easy to see that in all but the case of the identity operator that if a $callProc$ event were to propagate then information would be lost. For example, suppose $P = a \to P \;\Box\; R$ and $R = b \to R$. As the call to R on the right of the external choice is replaced by a $callProc$ event, it follows that the recursion would resolve the external choice which is clearly incorrect.

3.3. Formalising the Solution

We now consider how to formalise the solution sketched above and, in particular, we prove that the process that results from the transformations is equivalent, in a sense we later formalise, to the original simulation. In order to prove this, we firstly formally define what we are actually simulating. In particular, we define a *syntactic process* which corresponds, essentially, to a process definition in a CSP$_M$ file.

Definition 3.1. A *syntactic process* P is defined by the grammar $P ::= Op(P, \ldots, P) \mid N$, where N represents a *process name* and Op is a CSP-like operator. A *process environment* Γ is a function from process names to syntactic processes. For simplicity, we assume that $\Gamma(N)$ is never a process name, and is thus always of the form $Op(\ldots)$[3].

Given a syntactic process, it is possible to construct a simulation of it using the processes from Section 2.1. Thus, we define a function $sim^{\Gamma}(P)$ that, given a syntactic process P, returns the simulation of it. For ease, we assume that the $Operator$ process defined in Section 2.1 can be given a single vector of processes, rather than two vectors of **on** and **off** processes, respectively.

Definition 3.2. The *simulation of P in* Γ, denoted $sim^{\Gamma}(P)$ is defined recursively by:

$$\begin{aligned} sim^{\Gamma}(Op(P_1, \ldots, P_N)) &\mathrel{\widehat{=}} Operator(Op, \langle sim^{\Gamma}(P_1), \ldots, sim^{\Gamma}(P_N)\rangle) \\ sim^{\Gamma}(N) &\mathrel{\widehat{=}} N \end{aligned}$$

Note that the above simulation cannot be compiled by FDR if it contains recursive processes, as discussed above. However, it is strongly bisimilar to the original process, given Roscoe's results.

We now consider how to formalise the recursion refactorings we sketched in the preceding section. In order to formalise them, we need to distinguish between the different ways that an operator can use the events performed by its arguments. For example, $\Box$ only looks at the initial visible events performed by each of its arguments, whilst $|||$ looks at all the events performed by its arguments. We formalise this difference in the following definition.

Definition 3.3. An **on** process argument P of a CSP-like operator Op is *1-required* iff P can perform a visible event, i.e. there exists an inductive rule of the form:

$$\frac{\ldots \wedge P \xrightarrow{a} P' \wedge \ldots}{Op(\ldots, P, \ldots) \xrightarrow{b} Op'(\ldots).}$$

where $a \in \Sigma$. An argument P of a CSP-like operator Op is $(k+1)$*-required* iff there exists an inductive rule of the above form to a CSP-like operator $Op' \neq Identity$ such that P' is k-required for Op'. An argument P of an operator Op is *infinitely recursive* iff P is k-required for every $k \in \mathbb{N}$. An argument P of an operator Op is *only k-recursive* iff P is k-required, but not $k+1$-required.

For example, considering CSP: both arguments of $\|$ are infinitely recursive; both arguments of $\Box$ are only 1-required; the left argument of $\triangle$ is infinitely recursive but the right argument is only 1-required; the left argument of Θ_A is infinitely recursive but the right is not required as it is **off**. Note that, by definition of an **on** argument each **on** argument is at least 1-required.

[3] This is not a restriction in the tool, but is added only to simplify the presentation.

Using the above definitions, we can now formalise the recursion refactorings that we sketched in Section 3.2. Firstly, we change every recursive call[4] to a process N to be a $callProc$ event. Further, if an argument of an operator is infinitely recursive, we apply $WrapThread$, to ensure that no $callProc$ events propagate out. We formalise this as a function that returns a CSP process as follows.

Definition 3.4. Let Γ be a syntactic process environment and P a syntactic process. $rec\_sim^{\Gamma}(P)$ is defined inductively by:

$$
\begin{aligned}
rec\_sim^{\Gamma}(Op(P_1, \ldots, P_N)) &\mathrel{\widehat{=}} Operator(Op, \langle irec^{\Gamma}(Op, 1, P_1), \ldots, irec^{\Gamma}(Op, N, P_N)\rangle) \\
rec\_sim^{\Gamma}(N) &\mathrel{\widehat{=}} callProc.N \to STOP \\
irec^{\Gamma}(Op, i, N) &\mathrel{\widehat{=}} \begin{cases} WrapThread(rec\_sim^{\Gamma}(\Gamma(N))) & i \text{ is infinitely recursive} \\ rec\_sim^{\Gamma}(P) & \text{otherwise} \end{cases} \\
irec^{\Gamma}(Op, i, P) &\mathrel{\widehat{=}} rec\_sim^{\Gamma}(P)
\end{aligned}
$$

Note that the above simulation will no longer be strongly bisimilar to the original process (or indeed to $sim^{\Gamma}(P)$). This is because whenever a recursion occurs, there will be two extra τ's resulting from the hidden $callProc$ events (cf. $WrapThread$). Whilst this is of theoretical importance, this is not of any consequence to the tool, since all CSP denotational models equate P and $\tau \to P$. Hence, a particular refinement check would hold on the original process iff it holds on the above simulation.

Note that the above simulation is not sufficient for every CSP-like operator, nor every syntactic process. Thus, we now consider what syntactic processes the above simulation can handle. Firstly, observe that the process $P = Q \;\square\; P$ cannot be simulated, since the available transitions of P directly depend on those of P. Prohibiting such processes is an entirely reasonable restriction: such a P is not well-defined and would be rejected by any CSP-based tool (including FDR). Also, consider the process $P = a \to P \;|||\; STOP$. Again, the simulation will produce a process that cannot be compiled by FDR, since the left-hand argument is changed to $WrapThread(a \to callProc.P \to STOP)$. Hence, when FDR compiles P, after an a is performed another copy of P, itself containing another copy of $WrapThread$, will be spawned by the original $WrapThread$. We claim that prohibiting such processes is entirely reasonable, on the grounds that FDR cannot compile any such process either.

1-required arguments require extra restrictions to be placed since, by definition, no recursions can be permitted before a visible event has been performed. For example, the process $P = (P \sqcap STOP) \;\square\; a \to STOP$ is not well-defined since the τ of the $\sqcap$ does not resolve the $\square$, and thus the transitions of P again directly depend on those of P. We believe that such a restriction is reasonable, since FDR would also fail to compile such a process.

More problematically, any process of the form $P = Q \;\square\; a \to STOP$ cannot be simulated since the Q will be converted to $callProc.Q \to STOP$, but $callProc$ events are not allowed by the simulation of $\square$. Thus, it follows that the above simulation cannot deal with *immediately* recursive arguments. We extend the simulation to support such arguments in Section 3.4. We can formalise the set of processes for which the simulation is correct as follows. In the following definition, we only consider operators where each argument of the operator is either only 1-required or infinitely recursive, for simplicity. The definition and subsequent results can be generalised to arguments that are only k-required for $k > 1$.

[4] We don't actually need to change every recursive call; it actually suffices to change only recursive calls by non-recursive processes to recursive processes or by recursive processes to other recursive processes. We do not formalise this version since it adds extra complications.

Definition 3.5.

- Given a syntactic process P in a process environment Γ, P *recurses to* N iff either $P = N$, or $P = N'$ where $\Gamma(N')$ recurses to N, or $P = Op(P_1, \ldots, P_N)$ and one of the P_i recurses to N.
- A syntactic process P *recurses to N through only **on** arguments* iff $P = N$, or $P = N'$ and $\Gamma(N')$ recurses to N through only **on** arguments, or $P = Op(P_1, \ldots, P_N)$ and there exists an **on** P_i such that P_i recurses to N through only **on** arguments.
- A syntactic process P *has no named **on** arguments* iff P is not a name, and if $P = Op(P_1, \ldots, P_N)$ then each **on** P_i has no named **on** arguments.
- A syntactic process P *has 1-guarded recursion to* N iff P does not recurse to N, or $P = N'$ and $\Gamma(N')$ has 1-guarded recursion to N, or $P = Op(P_1, \ldots, P_N)$ and whenever $Op(P_1, \ldots, P_N) \xrightarrow{\tau} Op'(\mathbf{P})$, $Op'(\mathbf{P})$ has 1-guarded recursion to N.
- A process environment Γ *has correct simple recursion* iff for every $(N, P) \in \Gamma$, for every instance of $Op(P_1, \ldots, P_N)$ within P, and for each **on** P_i:
  - $*$ If P_i is infinitely recursive, then P_i does not recurse to N;
  - $*$ If P_i is only 1-required, then P_i does not recurse to N through only **on** arguments, P_i has no named **on** arguments and P_i has 1-guarded recursion to N.

Since the definition of *has correct simple recursion* only mentions operator arguments that are only 1-required and infinitely recursive, it follows that we can only allow operators that have only such arguments. With this observation in mind, we can state and prove our main result. Note in the following, as discussed after Definition 3.4, we weaken the equivalence from strong bisimulation to equivalence in any CSP denotational model.

Theorem 3.6. Let P be a syntactic process and Γ be a syntactic process environment with correct simple recursion. If every argument of every operator used in Γ or P is either infinitely recursive or only 1-required, then $WrapThread(rec\_sim^{\Gamma}(P)) = sim^{\Gamma}(P)$, where equality is interpreted in any CSP model.

Proof (Sketch). Let P and Γ be as per the lemma. Formally, this theorem can be proven by a structural induction over P. The correctness of this result follows primarily from the requirement that Γ has correct simple recursion. This means that, if an argument wishes to perform a $callProc$ event, then the operator must have evolved into the $Identity$ operator state, which allows the $callProc$ event to propagate, which can then be picked up by $WrapThread$. □

3.4. Immediate Recursions

The above solution enables many recursive process definitions to be compiled, but does not cover every possibility. As noted above, it does not correctly simulate processes that have *immediate* recursion, such as $P = a \to P \,\Box\, R$ and $R = b \to R$. In particular, in $rec\_sim^{\Gamma}(P)$ the right hand side of the external choice of P would be replaced by a $callProc$ event. However, as these are not allowed to propagate through the external choice it follows that only the choice of the left hand side would be presented.

The simplest solution to the above would be to *inline* the definition of R within P to yield the process $P = a \to P \,\Box\, b \to R$, which could be successfully simulated (in particular, it has correct simple recursion and thus Theorem 3.6 applies). However, this will not work on processes such as $P = a \to (P \,\Box\, b \to P)$ since, after inlining, the process still contains an incorrect usage of external choice (in this case, the inlined version would be $P = a \to ((a \to (P \,\Box\, b \to P) \,\Box\, P))$. To solve the above we introduce a fresh name R for the process $P \,\Box\, b \to P$. We then rewrite the definitions as:

$$P \mathrel{\widehat{=}} a \to R$$

$$R \mathrel{\widehat{=}} a \to R \,\Box\, b \to P.$$

Note that the above processes have no disallowed immediate recursions.

In general, if an argument of a process is only k-required, we need to inline sufficiently to ensure that none of the first k events can ever be $callProc$ events. If we do this, then it follows that the resulting syntactic process will satisfy the preconditions of Theorem 3.6 and thus this can be applied to correctly simulate the process. In order to formalise this we firstly expand the definition of has correct simple recursion to allow immediate recursion. As before, we concentrate only on operators where each argument is either infinitely recursive, or only 1-required. Again, the definition could be generalised as required.

Definition 3.7. A process environment Γ *has correct recursion* iff for every $(N, P) \in \Gamma$, for every instance of $Op(P_1, \ldots, P_N)$ within P, and for each **on** P_i:

1. If P_i is infinitely recursive, then P_i does not recurse to N;
2. If P_i is only 1-required, then P_i does not recurse to N through only **on** arguments and P_i has 1-guarded recursion to N.

Note that the second clause prohibits syntactic processes where the events for a process depend recursively on the process itself. Clearly such processes are not well-defined. We now define how to inline process definitions, in order to support the refactoring that was sketched above. Note that the following transformation is only well-defined if the process environment has correct recursion.

Definition 3.8. Given a syntactic process environment Γ', the *inlined process environment* Γ' is defined on each syntactic process contained in any process in Γ. In particular, $dom(\Gamma')$ is the set of all $N_{P'}$ where P' is a subprocess of some $P \in img(\Gamma)$. Further, Γ' is defined by:

$$\begin{aligned}
\Gamma'(N_{Op(P_1,\ldots,P_M)}) &\mathrel{\widehat{=}} Op(inline(Op, 1, P_1), \ldots, inline(Op, N, P_N)) \\
\Gamma'(N_{N'}) &\mathrel{\widehat{=}} \Gamma'(\Gamma(N')) \\
inline(Op, i, P_i) &\mathrel{\widehat{=}} \begin{cases} N_{P_i} & i \text{ is } \textbf{off} \\ \Gamma'(N_{N'}) & i \text{ is } \textbf{on} \text{ and } P_i = N' \\ \Gamma'(N_{Op'(\mathbf{P'})}) & i \text{ is } \textbf{on} \text{ and } P_i = Op'(\mathbf{P'}) \end{cases}
\end{aligned}$$

Note that the second clause of Γ' is well-defined by the assumption in Definition 3.5 that $\Gamma(N) \neq N'$. Further, observe that the $inline$ is semantics-preserving since it is essentially just a renaming operation. Thus, if $N \in dom(\Gamma)$, then $sim^{\Gamma}(N)$ is strongly bisimilar to $sim^{\Gamma'}(\Gamma'(N))$.

Clearly, the inlining process will only terminate if no **on** argument recurses back to itself. This is implied by the definition of has correct recursion in Definition 3.7.

Theorem 3.9. Let Γ be a process environment with correct recursion and let Γ' be the flattened process environment of Γ. Then Γ' has correct simple recursion.

Proof. Let Γ and Γ' be as per the lemma. By definition of correct recursion and correct simple recursion it suffices to show: for each $(N, P) \in \Gamma'$, each $Op(P_1, \ldots, P_N)$ that is a subprocess of P, and each **on** argument P_i, that P_i has no named **on** arguments. This follows immediately from the definition of $inline$. □

For optimisation reasons, the tool does not implement the refactoring exactly like this, but instead detects when the recursion refactoring has to be applied and does so only to such processes. This is discussed further in Section 4.

3.5. Limitations

The recursion refactorings that we have discussed above allow a large class of recursive processes to be successfully simulated. However, there are fundamental limitations to the range of operators that can be simulated in such a way. One good example of an operator that cannot be simulated using the above construction is the new CSP operator *synchronising external choice*, $\square_A$ [11]. This operator behaves likes a hybrid of external choice and generalised parallel. In particular, $P \square_A Q$ is resolved by P or Q performing a visible event outside of A, whilst if P or Q wishes to perform an event from A, it must synchronise with the other process. Thus, $\square_{\{\}}$ is equivalent to $\square$, whilst $\square_\Sigma$ is equivalent to $\underset{\Sigma}{\parallel}$.

This operator is particularly problematic for the recursion refactorings since it is the only CSP operator where an argument that is infinitely recursive can also be recursed through. For example, consider the process: $P = a \to b \to P \square_{\{a\}} a \to STOP$. In this process, P recurses through the first argument of $\square_{\{a\}}$, which is infinitely recursive[5], (cf. Definition 3.3). However, FDR has no problem compiling the above process, because when the b is performed, $\square_{\{a\}}$ is discarded, meaning that a transition back to P is found, as required.

There is no obvious way of altering the simulation to support an operator that might sometimes recurse (and thus requires the $callProc$ event to not be consumed by a $WrapThread$ inside $\square_{\{a\}}$), but sometimes never recurses (and thus requires that all $callProc$ events *must* be consumed by a $WrapThread$ inside $\square_{\{a\}}$). In [1] Roscoe actually conjectures that it is impossible to simulate $\square_A$ in such a way that FDR can successfully compile processes that use it. Thankfully, such operators are sufficiently uncommon that the simulation we provide is still applicable to a very wide variety of operational semantics.

4. Tool Support

As part of this work we have constructed a tool called tyger, written in Haskell, that automatically constructs the simulation and refactors the user's process definitions, as above. The input to tyger consists of two input files, the first of which specifies the operational semantics of the language. For example, the machine-readable version of the SOS rules for the CSP exception operator, as given in Section 1.1, can be defined as follows:

```
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
```

In the above, `Exception(P : InfRec, Q, A)` specifies that the exception operator takes 3 arguments, the first of which is a process that is infinitely recursive (cf. Definition 3.3). The `Syntax` line provides the information that tyger requires to parse the file. In particular, it specifies that the operator is a binary operator, with concrete syntax “`[| $3 |>`” where `$3` refers to the third argument (i.e. `A`), is non-associative and has a precedence of 12 (which is used to disambiguate non-bracketed statements). Using this line tyger dynamically constructs a parser (using the Haskell Parsec library[6]) to parse this file. The two `Rule` constructs

[5] Hence, this violates the conditions of has correct recursion, since P recurses through an infinitely-recursive argument.

[6] `http://hackage.haskell.org/package/parsec`

are precisely the machine readable versions of the SOS rules from Section 1.1 (noting that `diff(Sigma, A)` is the machine readable form of $\Sigma \setminus A$). Note that tau promotion rules are omitted as they are automatically added.

After parsing the above file, tyger performs a sort of type-checking on the operational semantics and infers which arguments of the operators are **off** and **on**. The type-checking is necessary because the side conditions of the operational semantic conditions (e.g. the `a <- Sigma`) can be written in a simple functional language, consisting of a simple set comprehensions. It also checks that the operators are CSP-like by checking a number of conditions on each SOS rule. For example, it checks that no **on** arguments are cloned, no **on** arguments are suspended, amongst several other conditions. The operational semantics are then converted into the form specified in Section 2.1. At this point, the simulation script can be produced, which is done by concatenating some constant code that implements the $Operator$ process with some generated code for the $Rules$ function.

The second input file contains process definitions which, thanks to the dynamic parser construction, can be specified using natural syntax. For instance, when defining a CSP process one can simply write `P = a -> P [] b -> Q`. Further, the script may contain definitions using the functional portion of the CSP$_M$ language, such as `head(<x>^xs) = x`. The script is then type-checked (using an experimental CSP$_M$ type checker), and then has the recursion refactorings applied to it. In particular, the inferred types are used to identify all processes within a file (note that the processes here are essentially the syntactic processes of Definition 3.5). Then, a call graph of the processes is constructed and the strongly connected components (SCCs) are deduced. This enables the recursive processes to be identified, which then allows the recursion refactorings to be run.

If the recursion refactoring specified in Section 3.4 were applied exactly as specified, this would result in many more named processes than are strictly necessary, slowing down the simulation. Thus, instead the following recursion refactorings are applied instead:

- Every call from a recursive process to a recursive process is replaced by a $callProc$ event;
- Every call from a non-recursive process to a recursive process Q instead calls the wrapped version, $WrapThread(Q)$ (so that $callProc$'s are processed);
- Every infinitely recursive argument Q of an operator is replaced by $WrapThread(Q)$;
- Every argument that is only 1-required is inlined as per Section 3.4.

Further, in order to define the $WrapThread$ process (cf. Section 3.2), the set of values that each recursive process can take has to be specified. For example, the following defines a recursive process P that takes a single integer value that can either be 0 or 1:

```
P :: ({0, 1}) -> Proc
P(x) = if x == 0 then a -> P(1) else b -> P(0)
```

Lastly, the transformed processes are pretty printed to a file. This file can be loaded into FDR and any assertions that are contained can be checked in the usual way.

Availability tyger has been open-sourced and is available from `https://github.com/tomgr/tyger`. Included with the code are several examples, including a full simulation of CSP, a simulation of Lowe's availability models [10], a simulation of Lowe's Readyness Testing extension to CSP [9], and a simulation of a portion of CCS [7].

5. Experiments

In order to demonstrate the effectiveness of the tool and to establish how efficient the simulation is, we performed a number of experiments which we now describe. In this section, all

	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

Table 2. Time in seconds to check if a CSP model of the Dining Philosophers is deadlock free for various numbers of philosophers.

	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

Table 3. Time in seconds to check if a CCS model of the Dining Philosophers is deadlock free for various numbers of philosophers. The CWB was unable to falsify the property within 30 minutes with 8 or more philosophers.

experiments were performed on a 64-bit Linux virtual machine with access to 2GB of RAM and two 2.2GHz cores. The simulation was run using a pre-release version of FDR3, running in single-threaded mode. CSP model checking was performed using the same pre-release version of FDR3 in single-threaded mode. CCS model checking was performed using version 7.1 of The Edinburgh Concurrency Workbench [12].

In order to directly measure the overhead of the simulation, we simulated CSP itself. Clearly, there is no point in doing this in general, but if we compare the time taken to check if a CSP process P satisfies a property to the time taken to check if the same property is satisfies by the simulation of P, then it follows that any difference must be due to the overhead of the simulation. Table 2 gives the time taken to check if a simple CSP model of the classic Dining Philosophers problem is deadlock free (in particular, it gives the time taken to find the first counterexample). Whilst the simulation is slower, note that the magnitude of the difference becomes less as the number of philosophers is increased. Further, the simulation is still reasonably performant and scales in an almost identical way to the non-simulated version, indicating that there should be no reason why larger models cannot be checked.

In order to demonstrate the that tool can effectively model other CSP-like process algebras, we compared the performance of a simulation of CCS [7] with a native CCS tool, The Edinburgh Concurrency Workbench [12] (henceforth, CWB). As noted in Section 1.2, CCS is actually not quite CSP-like, since τ resolves $+$ (which is the CCS analog of $\Box$). Hence, in order to simulate CCS we instead introduce a new event, *ccs_tau*, which is used instead of *tau* in the operational semantics rules. Then, explicit *ccs_tau* promotion rules are added to each operator and *ccs_tau* is hidden at the top-level of the simulation. It follows that the resulting operational semantics is CSP-like, although it is not compositional. Once this change has been made, the translation of the operational semantics into the form required for tyger is routine.

Table 3 details how long it took to verify if the Dining Philosophers problem is deadlock free for an increasing number of philsophers. Whilst CWB outperforms the simulation for 6 philosophers or less, for 7 philosophers or more the simulation actually outperforms CWB. Further, CWB was unable to falsify the property within 30 minutes when there were 8 or more philosophers whilst the simulation took just 54s to find a counterexample. To a large extent, this data justifies the motivation behind the tool. Not only is the simulation able to effectively simulate other process algebras, but is actually able to improve on the performance of custom tools (largely thanks to the optimisation performed on FDR).

We have had success in simulating a number of other CSP-like languages using tyger. One interesting example is Lowe's readyness testing extension to CSP [9]. This adds a new operator, "if ready a then P else Q", that tests if the event a is being offered and behaves like

P if so and like Q otherwise. Lowe demonstrated its usefulness by giving a notably simple solution to the readers and writers problem [13] that was fair to the writers. We were able to successfully simulate this, with the simulation taking 60s to complete a safety check.

In the author's experience, the performance of the simulation is perfectly adequate providing the number of events is resaonble, but unsuitable when the set of events become large. This is primarily because the alphabets used by the simulation become prohibitively large, and FDR appears to have trouble manipulating them.

6. Conclusions

In this paper we presented a CSP$_M$ adaptation of Roscoe's simulation that can be successfully used in FDR. Further, we gave a number of optimisations to Roscoe's simulation that ensure the resulting simulation runs reasonably efficiently. We then proved that these optimisations maintained the strong bisimulation between the original process and the simulation. We also provided some automated recursion refactorings that permit a large class of recursive definitions to be successfully compiled by FDR, even though the standard simulation could not be. We then sketched the construction of a tool that can produce the simulation automatically. Lastly, we described the results of several experiments that show the tool can effectively simulate other process algebras.

Future Work The overhead imposed by the simulation is essentially unavoidable and, therefore, there are unlikely to be ways of significantly improving the performance using the current approach. However, one option would be to add support directly within FDR for directly constructing the state machines for CSP-like languages. This would entirely avoid the overhead of the simulation and, further, would negate the need for the complicated recursion refactorings required.

There are several programs that produce CSP scripts as output, such as Roscoe's shared variable compiler [14] and Lowe's Casper Security Protocol Compiler [15]. The output of these tools can take a long time to compile, sometimes because it was not possible to express certain behaviours as a simple combination of CSP operators. Thus, it would be interesting to consider if domain-specific operators could be added to enable the above scripts to be expressed in a way that compiles more efficiently, using the new refinement checker mentioned in the previous paragraph.

Acknowledgements I would like to thank Gavin Lowe, who supervised a large part of this work as part of an Oxford undergraduate project [16]. Further, I would like to thank Gavin for reviewing an early draft of this paper and Bill Roscoe for many interesting discussions concerning this work. I would also like to thank the anonymous reviewers for providing many useful comments.

References

[1] A. W. Roscoe. On the Expressiveness of CSP. Draft of February 2011, available from `http://www.cs.ox.ac.uk/files/1383/complete(3).pdf`, 2011.

[2] A. W. Roscoe. CSP is Expressive Enough for Pi. In A.W. Roscoe C.B. Jones and K.R. Wood, editors, *Reflections on the work of C.A.R. Hoare*. Springer, 2010.

[3] C. A. R. Hoare. *Communicating Sequential Processes*. Prentice-Hall, Inc., Upper Saddle River, NJ, USA, 1985.

[4] A. W. Roscoe. *The Theory and Practice of Concurrency*. Prentice Hall, 1997.

[5] A. W. Roscoe. *Understanding Concurrent Systems*. Springer, 2010.

[6] Formal Systems (Europe) Ltd. *Failures-Divergence Refinement—FDR 2 User Manual*, 2011.

[7] R. Milner. *A Calculus of Communicating Systems*. Springer-Verlag New York, Inc., Secaucus, NJ, USA, 1982.

[8] R. Milner. *Communicating and Mobile Systems: the Pi-Calculus*. Cambridge University Press, June 1999.

[9] G. Lowe. Extending CSP with Tests for Availability. In *Communicating Process Architectures*, pages 325–347, 2009.

[10] G. Lowe. Models for CSP with Availability Information. In *EXPRESS'10*, pages 91–105, 2010. `http://dx.doi.org/10.4204/EPTCS.41.7`.

[11] P. Armstrong, G. Lowe, J Ouaknine, and A.W Roscoe. Model Checking Timed CSP. In *Proceedings of HOWARD (Festschrift for Howard Barringer)*. 2012.

[12] Faron Moller and Perdita Stevens. Edinburgh Concurrency Workbench user manual (version 7.1). Available from `http://homepages.inf.ed.ac.uk/perdita/cwb/`.

[13] P. J. Courtois, F. Heymans, and D. L. Parnas. Concurrent Control with "Readers" and "Writers". *Communications of the ACM*, 14(10):667–668, 1971.

[14] A. W. Roscoe and D. Hopkins. SVA, a Tool for Analysing Shared-variable Programs. In *Proceedings of AVoCS 2007*, pages 177–183, 2007.

[15] G. Lowe. Casper: A Compiler for the Analysis of Security Protocols. *Journal of Computer Security*, 6(1-2):53–84, 1998.

[16] Thomas Gibson-Robinson. Tyger: A Tool For Automatically Simulating CSP-Like Languages In CSP. MCompSci Thesis, University of Oxford, 2010.