# Proof Principles of CSP – CSP-Prover in Practice

Yoshinao Isobe[1] and Markus Roggenbach[2*]

[1] National Institute of Advanced Industrial Science and Technology, Japan,
y-isobe@aist.go.jp
[2] University of Wales Swansea, United Kingdom,
M.Roggenbach@Swan.ac.uk

**Abstract.** The process algebra CSP provides a well-established formalism for the modelling, analysis, and verification of concurrent systems. Besides being a specification language, CSP provides a valuable set of proof principles. We show in tutorial style, how these proof principles are made available in our tool CSP-Prover. Overall, CSP-Prover turns out to be an off-the-shelf proof tool ready for use in applications.

## 1 Introduction

The process algebra CSP [7, 16] provides a well-established, theoretically thoroughly studied, and in industry often applied formalism for the modelling and verification of concurrent systems. CSP has been successfully applied in areas as varied as distributed databases, parallel algorithms, train control systems [4], fault-tolerant systems [3], and security protocols [20].

Within the spectrum of models of concurrency, semantically CSP is based on the so-called *Hoare-Languages*. In the classification of [21], these are of type 'behaviour' (during a system's execution, a state can only appear once), 'interleaving' (parallel execution is expressed via non-determinism), and 'linear time' (CSP abstracts from the point of branching in non-deterministic choices). For such models of concurrency, CSP provides a rich variety of equivalences and refinements.

Fixing one syntax, CSP offers different semantical models, each of which is dedicated to special verification tasks. The traces model $\mathcal{T}$, e.g., covers *safety properties*. *Lifeness properties* can be studied in more elaborate models. *Deadlock analysis*, e.g., is best carried out in the stable-failures model $\mathcal{F}$, the failures-divergences model $\mathcal{N}$ allows for *lifelock analysis*, the stable-revivals model $\mathcal{R}$ [18] has recently been designed to study *responsiveness*. The analysis of *fairness properties* requires models based on infinite traces, see [16, 19] for further details.

The literature on process algebra, see, e.g., [2], has suggested a variety of proof principles for process algebra in general, which, naturally, are also available in the context of CSP. These proof principles provide a valuable tool set in order to

* This cooperation was supported by the EPSRC Project EP/D037212/1.

analyze, validate and verify concurrent systems. To a certain extend, the success of process algebra in general, and especially of Csp, is based on these very principles. The overall purpose of these techniques is to lift *semantical proofs* to *proofs on the syntactic level*: Process algebra is famous for its rich set of algebraic laws, which allow for the manipulation of specifications on the syntactic level.

Csp-Prover [8, 9, 11, 10] is an interactive proof tool for Csp based on the theorem prover Isabelle/HOL [14]. With its theorem proving approach Csp-Prover complements the established model checker FDR [13] as a proof tool for Csp. Csp-Prover is generic in the various Csp models. Currently, Csp-Prover fully supports the traces model $\mathcal{T}$ and the stable-failures model $\mathcal{F}$. Implementations of the failures-divergences model $\mathcal{N}$ and the stable-revivals model $\mathcal{R}$ are well under way.

In this paper we review proof principles of process algebra and show how they can be applied in Csp-Prover. It turns out that Csp-Prover provides a suitable platform to study and apply proof principles of Csp. However, sometimes subtle but important changes are required in order to obtain an 'automatized' version of an established proof technique, which so-far was carried out manually only.

The various Csp models are defined in terms of denotational semantics. Consequently, all syntactic proof principles such as algebraic laws, fixed point induction, or the existence and uniqueness of normal forms have to be proven to be correct with respect to a given denotational semantics. Csp-Prover provides an encoding of these denotational semantics as well as of a wide range of already proven to be correct syntactic proof principles. For the stable-failures model $\mathcal{F}$ the implemented principles actually are complete [10]. Overall, Csp-Prover can be used as an off-the-shelf proof tool, which easily can be extended by domain specific theorems.

Tools similar to Csp-Prover have been suggested by Tej/Wolff [24, 23] and Dutertre/Schneider [5, 22]. Here, Csp-Prover continues and extends the work by Tej/Wolf. Their tool HOL-Csp is focused on the failure-divergences model $\mathcal{N}$. While Csp-Prover works with a deep encoding of the language, HOL-Csp provides a shallow encoding of Csp in Isabelle/HOL. Consequently, in HOL-Csp proofs principles related to the Csp syntax are not easy to realise. Schneider/Dutertre base their tool on the theorem prover PVS. Their work is tailored to the verification of security protocols. Their prover implements a restricted version of traces model $\mathcal{T}$.

The paper is organized as follows: First we give an overview on the process algebra Csp as it is available in Csp-Prover. Then we show how to prove and how to apply *algebraic laws* in Csp-Prover. Section 4 discusses recursively described systems, for which *fixed-point induction* provides a powerful and often adequate syntactical proof principle. Finally, we discuss *deadlock analysis*: In the context of Csp, deadlock-freedom is often proved by means of abstraction. An alternative is to look into specialized theorems that cover deadlock-free networks.

| | | |
|---|---|---|
| $P ::=$ | $p(x_1, \ldots, x_n)$ | %% (parametrized) process name |
| | $\mid$ *Skip* | %% successfully terminating process |
| | $\mid$ *Stop* | %% deadlock process |
| | $\mid$ *Div* | %% divergence |
| | $\mid$ $a \rightarrow P$ | %% action prefix |
| | $\mid$ $P \,\Box\, P$ | %% external choice |
| | $\mid$ $P \sqcap P$ | %% internal choice |
| | $\mid$ $? \, x : A \rightarrow P(x)$ | %% prefix choice |
| | $\mid$ $! \, s : S \bullet P(s)$ | %% replicated internal choice |
| | $\mid$ *if b then P else P* | %% conditional |
| | $\mid$ $P \,\|\|\|\, P$ | %% interleaving |
| | $\mid$ $P \,\|\, P$ | %% synchronous parallel |
| | $\mid$ $P \,[\![ X ]\!]\, P$ | %% generalized parallel |
| | $\mid$ $P \,\fatsemi\, P$ | %% sequential composition |
| | $\mid$ $P \setminus X$ | %% hiding |
| | $\mid$ $P[[r]]$ | %% relational renaming |
| | $\mid$ $P \downarrow n$ | %% depth restriction |

**Fig. 1.** Syntax of basic CSP processes in CSP-Prover.

## 2 The process algebra CSP in CSP-Prover

Fig. 1 shows the *basic* CSP *processes* available in CSP-Prover. These processes are defined relatively to an alphabet of actions $\Sigma$ and a set of process names. A process can be a (parametrized) process name or one of the primitive processes *Skip*, *Stop*, and *Div*. Actions can prefix a process. The choice between processes can be external or internal. Both choice operators are also available in a generalized version: the prefix choice operator allows the environment to choose between actions in a subset $A$ of the alphabet $\Sigma$; in the replicated internal choice, the process chooses an index $s$ from a set $S$ of selectors, where this set of selectors $S$ is either a set of natural numbers or a set of subsets of the alphabet $\Sigma$. The further progress of a process can depend on a condition. Various parallel operators are available, differing in their synchronization condition: with interleaving the processes run independent of each other; synchronous parallel forces the processes to run in lock-step; the general parallel operator synchronizes the processes in the subset $X$ of the alphabet $\Sigma$. There is hiding and renaming. Finally, it is possible to restrict a process to its first $n$ actions. Replicated internal choice is described in [10], otherwise see [16] for a detailed discussion of these operators.

In the setting of CSP-Prover, basic processes are terms of the type

```
('a,'b) proc
```

where `'a` is an Isabelle type variable to be instantiated with the type of process names, and, similarly, `'b` is an Isabelle type variable to be instantiated with the alphabet of actions. For example, the basic processes $a \rightarrow$ *Skip* and $?x : \{a, b\} \rightarrow P(x)$ can be typed-checked in Isabelle by declaring them as a `term`:

```
term "a -> SKIP"
term "? x:{a,b} -> P(x)"
```

Isabelle gives both terms the type `('a,'b) proc`.

A Csp *specification* consists of a set of *equations* of the form

$$p(x_1, \ldots, x_k) = P$$

where $p(x_1, \ldots, x_k)$ is a process name with parameters $x_1, \ldots, x_k$ and $P$ is a basic process. A typical example is the following Csp specification of a buffer with capacity one:

$$\begin{aligned} \mathit{Empty} &= \mathit{left}?x \rightarrow \mathit{Full}(x) \\ \mathit{Full}(x) &= \mathit{right}!x \rightarrow \mathit{Empty} \end{aligned}$$

Such a system of equations can be seen as a mapping from process names to basic processes. This is the way, how Csp-Prover encodes these equations, see Fig. 2. Our Csp specification `myBuffer` is declared as a function of type

```
myNames => (myNames, myAlphabet) proc
```

(line 8), the equations are encoded using the Isabelle command `primrec` (line 10 & line 11). The operator `$` is a type constructor which converts process names into basic processes.

Note that the specification in Fig. 2 describes a *generic* buffer. The type `data` is not further specified and can be instantiated with respect to the particular needs of an application. Semantically, `data` stands for an arbitrary set of values. Observe further that the equation in line 11 takes a variable `x` of type `data` as a parameter. This means that in the case that `data` is instantiated with an infinite set, the buffer specification essentially consists of infinitely many equations. Csp-Prover provides the means to analyze both generic Csp specifications as well as Csp specifications consisting of infinitely many equations.

The Csp-Prover User Guide, available at [8] discusses the full concrete syntax available.

## 3 Algebraic Laws

Algebraic laws are at the core of process algebra. They form the basic building block for all advanced techniques of process algebraic reasoning over concurrent systems. In this section we show how to prove algebraic laws and how to apply them in Csp-Prover.

In the context of Csp, algebraic laws come in two flavours, namely as laws on equality and as laws on refinement. Here, we consider laws on equality only. Laws on refinement can be treated in the same way as laws on equality. This has its reason in the following equivalence, which in general holds in the various Csp models:

$$P \sqsubseteq Q \Leftrightarrow P = P \sqcap Q$$

I.e., refinement can be expressed in terms of equality. In this sense, refinement can be considered as syntactic sugar, although in practice refinement surely is

```
1  (* data *)
2    typedecl data
3    datatype myAlphabet = left data | right data
4    datatype myNames = Empty | Full data
5
6  (* process *)
7    consts
8      myBuffer :: "myNames => (myNames, myAlphabet) proc"
9    primrec
10     "myBuffer (Empty) = left ? x -> $(Full x)"
11     "myBuffer (Full x) = right ! x -> $(Empty)"
```

**Fig. 2.** A script for the process *Buffer* in Csp-Prover

a notion of its own right. For this reason, Csp-Prover also offers specific proof procedures for refinement.

The manual verification of process algebraic laws with respect to a given non-algebraic semantics is an important but error prone and complex task. Errors found in algebraic laws years after their publication demonstrates the need for mechanised theorem proving. The latest example in the Csp context is the correction of two step laws, which, published in 1998, were shown to be incorrect in 2006, see [10]. With the exception of the stable failures model $\mathcal{F}$, for most Csp models it is an open question if they have a complete axiomatic semantics. Thus, the proof practice might require the proof of new, yet not considered algebraic laws. Section 3.1 presents – in the stable failures model $\mathcal{F}$ – a typical proof of a process algebraic law in Csp-Prover and explains the basic techniques involved.

Many tasks in analyzing concurrent systems with process algebra can be carried out solely by applying algebraic laws. Csp-Prover offers a rich collection of such laws, which have already proven to be correct. In Section 3.2, we demonstrate – this time in the traces model $\mathcal{T}$ – how Csp-Prover allows one to algebraically reason about systems in a natural and intuitive way. It turns out that after structuring the proof into reasonably small subproblems, Csp-Prover's tactics can automatically discharge the arising proof obligations. Due to the sheer number – about 80 laws are required to completely capture the stable failures model [10] – as well as of the complexity of these algebraic laws – the $\Box$–step law studied in Section 3.1 below is of 'medium' complexity only – Csp and, we believe, process algebra in general require an automated reasoning approach.

Comparing the complexity of the two proof approaches demonstrated, namely first carrying out a proof on the denotational semantics of Csp and then arguing on a system using process algebraic laws only, the proof on the syntactic level turns out to be much simpler although it deals with the more involved equation.

### 3.1 Correctness proofs of algebraic laws

As an example of a semantical proof we consider the law '$\Box$–step' concerning the external choice operator:

$$(?\,x : A \to P(x)) \,\Box\, (?\,x : B \to Q(x)) \;= ?\,x : (A \cup B) \to$$
$$(\mathit{if}\ (x \in A \cap B)\ \mathit{then}\ P(x) \sqcap Q(x)\ \mathit{else\ if}\ (x \in A)\ \mathit{then}\ P(x)\ \mathit{else}\ Q(x)).$$

This law studies the behaviour of a process, which is obtained by combining the two processes $(?\,x : A \to P(x))$ and $(?\,x : B \to Q(x))$ with the external choice operator. The rhs of $\Box$–step captures which actions are possible in the first step, and – depending on this first step – how the combined process behaves further. First, the combined process makes all actions $x \in (A \cup B)$ available to the environment. After the environment has chosen which $x$ the combined process should perform, the behaviour of the combined process depends on the set to which this $x$ belongs: if $x$ is in the intersection of $A$ and $B$, then the environment has actually made no choice between the two possible branches, and the originally external choice becomes an internal choice between $P(x)$ and $Q(x)$; if $x$ belongs to $A$ but not to $B$, then $P(x)$ is executed; if $x$ belongs to $B$ but not to $A$, then $Q(x)$ is executed. In Csp, laws such as $\Box$–step are known as 'step laws'. These step laws capture essential algebraic properties of the various operators: There is a step law for every Csp operator; the step laws hold in all (main) Csp models.

In the following, we will use Csp-Prover to prove that $\Box$–step holds in the stable failures model $\mathcal{F}$. The denotational semantics of $\mathcal{F}$ maps process expressions via functions *traces* and *failures* to pairs

$$(T, F), \quad \text{where } T \subseteq \Sigma^{*\checkmark} \text{ and } F \subseteq \Sigma^{*\checkmark} \times \mathcal{P}(\Sigma^{\checkmark}).$$

Here, $\Sigma$ is the alphabet of actions from the syntax definition of Csp [3].

To prove the law $\Box$–step, we need to show: for all choices of $\Sigma$, for all choices of $A \subseteq \Sigma$ and $B \subseteq \Sigma$, and for all interpretations of the basic processes $P(x)$ and $Q(x)$ in the semantic domain of $\mathcal{F}$ we have:

$$\mathit{traces}(\mathit{Ext}_{(A,P,B,Q)}) = \mathit{traces}(\mathit{Step}_{(A,P,B,Q)}) \qquad (\#1)$$
$$\mathit{failures}(\mathit{Ext}_{(A,P,B,Q)}) = \mathit{failures}(\mathit{Step}_{(A,P,B,Q)}) \qquad (\#2)$$

where $\mathit{Ext}_{(A,P,B,Q)}$ ($\mathit{Step}_{(A,P,B,Q)}$) denotes the lhs (rhs) of the equation $\Box$–step. In the following, we consider the proof of (#2) only:

In a *manual proof*, we first have to compute the sets $\mathit{failures}(\mathit{Ext}_{(A,P,B,Q)})$ and $\mathit{failures}(\mathit{Step}_{(A,P,B,Q)})$ by applying the semantic clauses of the model $\mathcal{F}$.

[3] $\Sigma^{\checkmark} = \Sigma \cup \{\checkmark\}$, $\Sigma^{*\checkmark} = \Sigma^{*} \cup \Sigma^{*} \frown \langle\checkmark\rangle$, where $\checkmark$ is a special symbol denoting successful termination, see [16] for a detailed discussion of $\mathcal{F}$.

```
1   lemma cspF_Ext_choice_step:
2      "(? x:A -> P(x)) [+] (? x:B -> Q(x)) =F[M,M]
3      ? x:(A Un B) -> (IF (x : A Int B) THEN P(x) |~| Q(x)
4                          ELSE IF (x : A) THEN P(x) ELSE Q(x))"
5   apply (simp add: cspF_cspT_semantics)
6   apply (simp add: cspT_Ext_choice_step)
7   apply (rule order_antisym, auto)
8   (* ⊆ *)
9      apply (simp add: in_traces in_failures)
10     apply (auto)
11  (* ⊇ *)
12     apply (simp add: in_traces in_failures)
13     apply (elim disjE conjE exE, force+)
14     apply (case_tac "a : B", simp_all add: in_failures)
15     apply (case_tac "a : A", simp_all add: in_failures)
16  done
```

**Fig. 3.** A proof script for step law of the external choice

After some simplifications we obtain:

$$
\begin{aligned}
&\mathit{failures}(\mathit{Ext}_{(A,P,B,Q)}) = \\
&\quad \{(\langle\rangle, X) \mid A \cap X = \emptyset\} \cup \{(\langle\rangle, X) \mid B \cap X = \emptyset\} \cup \\
&\quad \{(\langle a\rangle \frown t, X) \mid a \in A \wedge (t, X) \in \mathit{failures}(P(a))\} \cup \\
&\quad \{(\langle a\rangle \frown t, X) \mid a \in B \wedge (t, X) \in \mathit{failures}(Q(a))\} \qquad (\#3) \\
&\mathit{failures}(\mathit{Step}_{(A,P,B,Q)}) = \\
&\quad \{(\langle\rangle, X) \mid (A \cup B) \cap X = \emptyset\} \cup \\
&\quad \{(\langle a\rangle \frown t, X) \mid a \in A \cup B \wedge \\
&\qquad\qquad \mathit{if}\ (a \in A \cap B)\ \mathit{then}\ (t, X) \in \mathit{failures}(P(a)) \cup \mathit{failures}(Q(a)) \\
&\qquad\qquad \mathit{else\ if}\ (a \in A)\ \mathit{then}\ (t, X) \in \mathit{failures}(P(a)) \\
&\qquad\qquad \mathit{else}\ (t, X) \in \mathit{failures}(Q(a))\} \qquad (\#4)
\end{aligned}
$$

Using standard arguments on sets, one can show that the sets (#3) and (#4) are indeed equal and that therefore the step law $\Box$–step holds in $\mathcal{F}$.

In contrast to this approach, Fig. 3 shows a *proof-script* in Csp-Prover for this step-law. First, we state $\Box$–step to be our proof goal (line 1). The command in line 5 splits the main goal into two subgoals corresponding to the equations (#1) and (#2) above. The first subgoal (#1) is discharged at the line 6. Here, we use the lemma `cspT_Ext_choice_step` which belongs to the traces model $\mathcal{T}$. Now we deal with the second subgoal, the equation (#2). We prove this equality of sets by two subset relations, which we obtain as new subgoals at the line 7:

$$
\begin{aligned}
&\forall (t, X) \in \mathit{failures}(\mathit{Ext}_{(A,P,B,Q)}).\ (t, X) \in \mathit{failures}(\mathit{Step}_{(A,P,B,Q)}) \qquad (\#5) \\
&\forall (t, X) \in \mathit{failures}(\mathit{Step}_{(A,P,B,Q)}).\ (t, X) \in \mathit{failures}(\mathit{Ext}_{(A,P,B,Q)}) \qquad (\#6)
\end{aligned}
$$

The subgoal (#5) is proved in the lines 9 and 10, the subgoal (#6) is proved in lines 12–15. For both subgoals, the sets of failures[4] of the both processes, i.e. (#3) and (#4), are automatically derived in Csp-Prover, see the lines 9 and 12. This is a powerful technique. Deriving the denotations of processes according to

[4] The model $\mathcal{F}$ requires the trace sets in order to derive the failures of the external choice between processes.

the semantical clauses of a Csp model is a tedious but error prone and complex task – note that the sets (#3) and (#4) are simplified versions of the sets derived from the semantical clauses. The commands in line 10 and in the lines 13–15 finally check that (#3) and (#4) are equal.

### 3.2 Proofs based on algebraic laws

Given a set of algebraic laws such as $\Box$−step, how can we use these laws in Csp-Prover in order to reason about a Csp specification? Let us consider an example out of the context of testing from formal specifications, see [12]. Given the Csp specification of a system, and given a test in the form of a sequence of actions, shall we expect an implementation of this specification, the so-called system under test, to engage in this sequence of actions? More concrete, let the system be a simple calculator which reads values $x$ and $y$ from a keyboard and then shows on its display the sum $x + y$ of these values:

$$Calculator \ = ?x : Button \rightarrow ?y : Button \rightarrow \ Display!(x + y) \rightarrow Skip$$

Consider as a test the process which represents the sequence of pressing the button for zero, pressing the button for one, and then expecting the display to show a one:

$$Test = Button!0 \rightarrow \ Button!1 \rightarrow \ Display!1 \rightarrow Stop$$

In order to prove that *Test* is a sequence that any correct implementation of *Calculator* has to execute, we need to discharge several proof obligations, see [12] for the details. Here, we consider just one of the equations that need to be established in this context:

$$(((Calculator \parallel Test)\ [[MyRenaming]]) \ [\![ \{a\} ]\!] \ Count3) \setminus \{a\} =_{\mathcal{T}} OK \rightarrow Stop$$

Here, $MyRenaming = \{(x, a) \mid x \in \Sigma\}$ and $Count3 = a \rightarrow a \rightarrow a \rightarrow OK \rightarrow Stop$. This equation can be shown to be correct using the following lemmas:

1. $Calculator \parallel Test =_{\mathcal{T}} Test$ `(* parallel_one *)`
   The *Calculator* and the *Test* agree on the three actions prescribed by *Test*.
2. $Test[[MyRenaming]] =_{\mathcal{T}} a \rightarrow a \rightarrow a \rightarrow STOP$ `(* renaming *)`
   All these actions are renamed into $a$.
3. $a \rightarrow a \rightarrow a \rightarrow STOP \ [\![ \{a\} ]\!] \ Count3 =_{\mathcal{T}} Count3$ `(* parallel_two *)`
   *Count3* verifies that there are three agreed actions and communicates $OK$.
4. $Count3 \setminus \{a\} =_{\mathcal{T}} OK \rightarrow Stop$ `(* hiding *)`
   After hiding all the $a$'s, the action $OK$ remains the only one visible.

Fig. 4 shows how this proof idea can be formalized in Csp-Prover, where we assume – for the moment – that the above equations have already been proven. The tactic `cspT_simp_with_tac` takes care of the rewriting process. This tactic has as parameters the name of the lemma to be applied, e.g., `"parallel_one"`, and the number of the proof goal to which the lemma shall be applied – in

our case always the first goal. Using the four lemmas stated above, the proof script rewrites the lhs of the original equation till it is syntactically identical to the rhs, at which point the equation trivially holds. Note how this proof script corresponds directly to the natural line of argument shown above.

```
1   theorem
2   "(((Calculator || Test) [[ MyRenaming ]] ) |[{a}]| Count3) -- {a} =T Ok -> STOP"
3      apply(tactic { * cspT_simp_with_tac "parallel_one" 1 * } )
4      apply(tactic { * cspT_simp_with_tac "renaming" 1 * } )
5      apply(tactic { * cspT_simp_with_tac "parallel_two" 1 * } )
6      apply(tactic { * cspT_simp_with_tac "hiding" 1 * } )
7   done
```

**Fig. 4.** Discharging a proof obligation from formal testing

It remains to prove the four lemmas used in our theorem. These four lemmas can be proven following a systematic approach, which we demonstrate for the hiding lemma, see Fig. 5. CSP-Prover's tactic `cspT_hsf_tac` is usually able to prove simple equations. Adding a `+` to a proof command triggers its repeated execution till it fails. In order to prove the hiding lemma, we consider three basic cases: the communication is $a$, the communication is $OK$, and hiding is to be applied on the process *Stop*. Note that in `hide_a` we use the process name $P$ as a variable, which later can be instantiated with an arbitrary process.

```
1   lemma hide_a: "(a -> P) -- {a} =T P -- {a}"
2      apply (tactic { * cspT_hsf_tac 1* } )+
3   done
4
5   lemma hide_OK: "(OK -> P) -- {a} =T OK -> (P -- {a} )"
6      apply (tactic { * cspT_hsf_tac 1 * } )
7   done
8
9   lemma hide_STOP: "STOP -- {a} =T STOP"
10     apply (tactic { * cspT_hsf_tac 1 * } )
11  done
12
13  lemma hiding: "(a -> a -> a -> OK -> STOP) -- {a} =T OK -> STOP"
14     apply(tactic { * cspT_simp_with_tac "hide_a" 1 * } )+
15     apply(tactic { * cspT_simp_with_tac "hide_OK" 1 * } )
16     apply(auto)
17     apply(tactic { * cspT_simp_with_tac "hide_STOP" 1 * } )
18  done
```

**Fig. 5.** Some typical lemmas

For the situation that the available tactics fail to prove an equation CSP-Prover also offers proof commands that allow the user to take more detailed control of the rewriting process: Equations can be decomposed into their left and right hand side; equations can be glued together again; basic processes can be decomposed into their parts; specific algebraic laws can be chosen for the

rewriting process. These techniques allows the user to choose exactly the point at which an algebraic law shall be applied.

## 4 Fixed point analysis

In general, Csp specifications are recursive definitions. Take for instance the Csp specification

$$X = (a \to X) \,\Box\, X$$

which shall define the behaviour of a process $X$. Possible interpretations of $X$ are, for instance, the processes $Q_a$ and $Q_{ab}$ : [5]

$$\begin{aligned} Q_a &= (a \to Q_a) \\ Q_{ab} &= (a \to Q_{ab}) \,\Box\, (b \to Q_{ab}) \end{aligned}$$

Thus, the two standard questions on recursively defined objects apply to Csp specifications: Does there exist a solution? If there exists a solution, is this solution uniquely determined? In the context of analysing systems, also a third question arises: What proof principles can be applied to the solution?

For Csp, these questions are answered either in terms of order theoretic techniques based upon Tarski's fixed point theorem or in terms of metric techniques based upon Banach's fixed point theorem. The following tabular summarizes the structures available for the main Csp models, see [16, 18]:

| Model | order theoretic structure | metric structure |
|---|---|---|
| $\mathcal{T}$ | complete lattice | complete metric space |
| $\mathcal{N}$ | for finite alphabet $\Sigma$ : complete partial order | complete metric space |
| $\mathcal{F}$ | complete lattice | complete metric space |
| $\mathcal{R}$ | complete lattice | no published results |

With Tarski's fixed point theorem one can show the existence of a solution, however, there can be several ones. Depending on the model under consideration, Csp selects the smallest (respectively the largest) fixed point under the chosen ordering relation and achieves in this way uniqueness. In contrast to this, Banach's fixed point theorem yields the existence of a unique solution. Consequently, only order-theoretic reasoning can be applied to the specification above. The metric approach fails.

The applicability of Tarski's and Banach's fixed point theorem depends on the semantical properties of the given Csp specification. For Tarski's theorem, on the semantical level all functions are required to be continuous with respect to the chosen ordering relation. In most Csp models, all the standard Csp operators

[5] Replacing $X$ on the rhs by $Q_a$ yields with help of the idempotence law $P \,\Box\, P = P$: $(a \to Q_a) \,\Box\, Q_a = (a \to Q_a) \,\Box\, (a \to Q_a) = a \to Q_a = Q_a$. For $Q_{ab}$ we obtain: $(a \to Q_{ab}) \,\Box\, Q_{ab} = (a \to Q_{ab}) \,\Box\, (a \to Q_{ab}) \,\Box\, (b \to Q_{ab}) = (a \to Q_{ab}) \,\Box\, (b \to Q_{ab}) = Q_{ab}$.

and their composition yield continuous functions. Banach's theorem, however, requires the functions on the semantical level to be contracting. Here, the process algebraic literature has introduced the concept of 'guardedness' of a process expression. In Csp, a guarded process expression yields a contracting function on the semantical level. Thus, both semantical requirements, namely continuity and contractiveness, can be captured on a purely syntactical level. The process expression $(a \rightarrow X) \,\Box\, X$ fails to be guarded, the expressions $a \rightarrow Q_a$ and $(a \rightarrow Q_{ab}) \,\Box\, (b \rightarrow Q_{ab})$ are guarded.

Concerning the analysis of recursively defined objects, fixed-point-induction offers a powerful proof method. In order to prove, e.g., in the traces model $\mathcal{T}$

$$X =_{\mathcal{T}} Q_a$$

one needs to show

$$Q_a \sqsubseteq_{\mathcal{T}} (a \rightarrow Q_a) \,\Box\, Q_a \quad \wedge \quad X \sqsubseteq_{\mathcal{T}} a \rightarrow X \qquad (*)$$

i.e., $Q_a$ needs to be refined by the process which is obtained by substituting $X$ by $Q_a$ on the rhs of the defining equation of $X$, and $X$ needs to be refined by the process which is obtained by substituting $Q_a$ by $X$ on the rhs of the defining equation of $Q_a$. $Q_{ab}$ is not equivalent to $X$, as $X \not\sqsubseteq_{\mathcal{T}} (a \rightarrow X) \,\Box\, (b \rightarrow X)$. The two refinements $(*)$ can be proven as follows:

$$Q_a =_{\mathcal{T}} Q_a \,\Box\, Q_a =_{\mathcal{T}} (a \rightarrow Q_a) \,\Box\, Q_a, \qquad (*_1)$$

$$X =_{\mathcal{T}} (a \rightarrow X) \,\Box\, X \sqsubseteq_{\mathcal{T}} (a \rightarrow X) \,\Box\, \mathit{Stop} =_{\mathcal{T}} (a \rightarrow X) \qquad (*_2)$$

Here, we use the facts that *Stop* is the largest process with respect to the order $\sqsubseteq_{\mathcal{T}}$, and that $P = Q \Leftrightarrow P \sqsubseteq Q \wedge Q \sqsubseteq P$.

### 4.1 Basic fixed point analysis techniques in Csp-Prover

The above examples are complex enough to demonstrate basic techniques for fixed point analysis in Csp-Prover. The next section on deadlock analysis will provide more elaborate examples. Fig. 6 shows the encoding of the Csp specifications for $X$ and $Q_a$. `PNfun` is a *reserved word* of Csp-Prover. It is used to specify that for each process name $A$ the equation $A =$ `PNfun`$(A)$ holds. This function `PNfun` can be used for each type of process names by the option *overloaded*, as shown in lines 8 and 14.

Fig. 7 shows the proof script for $Q_a \sqsubseteq_{\mathcal{T}} X$ and $X \sqsubseteq_{\mathcal{T}} Q_a$ (i.e. $X =_{\mathcal{T}} Q_a$). First, we declare that the cpo approach is to be used in this proof (line 4)[6]. The refinement lemmas at lines 10 and 19 give the goals $(*)$ to Csp-Prover. It is necessary to explicitly specify the type of `$X` – otherwise `$X` has no information on the type `Event`. If this typing is left out, the goal becomes more general and is

[6] cpo - complete partial order. In contrast to this, it is possible to use to set `FPmode` to the value `CMSmode`; cms - complete metric space, i.e., one works with Banach's theorem.

```
1  theory xq = CSP_T:
2    datatype Event = a | b
3
4  (* unguarded process *)
5    datatype ungPN = X
6    consts ungfun :: "ungPN => (ungPN, Event) proc"
7    primrec "ungfun X = a -> $X [+] $X"
8    defs (overloaded) ungfun_def [simp]: "PNfun == ungfun"
9
10 (* guarded process *)
11   datatype gPN = Qa
12   consts gfun :: "gPN => (gPN, Event) proc"
13   primrec "gfun Qa = a -> $Qa"
14   defs (overloaded) gfun_def [simp]: "PNfun == gfun"
15 end
```

**Fig. 6.** Encoding of recursive CSP specifications

not valid any more. The proof strategy used in lines 11–13 is exactly the same as the one in ($*_1$) explained above. First, fixed point induction is applied to the rhs (line 11). The function `fx` explicitly states for each process name $A$ on the rhs, which process on the lhs is expected to be refined by $A$. The command in line 12 instantiates the process name $Q_a$. After this command, the subgoal `$Qa <=T a -> $Qa [+] $Qa` is displayed. This can be proven by the unwinding (lines 13) as stated above ($*_1$). The reverse direction $X \sqsubseteq_{\mathcal{T}} Q_a$ can be proven in a similar way (lines 15–22).

While, for mathematical reasons, fixed point induction of type cpo can be applied only for unfolding the rhs as demonstrated in Fig. 7, metric fixed point induction can also be applied on the lhs. In the proof practice this is of major advantage. In a refinement goal *spec* $\sqsubseteq$ *impl*, the process *impl* often has a concurrent structure like $(P1 \llbracket X \rrbracket P2) \setminus Y$, which cannot be unfolded.

The price to pay for metric fixed point induction, however, is that it requires process expression to be guarded. To this end, CSP-Prover provides a predicate `guardedfun` to check if a process expression `fun` is guarded. In our example, the application `guardedfun ungfun` leads to `False`, where `ungfun` is the function defined at line 7 in Fig. 6. This demonstrates how CSP-Prover prevents the use of metric fixed point induction for the proof goal $X =_{\mathcal{T}} Q_a$.

## 5 Deadlock analysis

Deadlocks are certainly the best known and also most feared failures exhibited by concurrent systems. In the analysis of concurrent systems, proof of deadlock-freedom is as fundamental as termination proofs are in the context of sequential systems.

In CSP, deadlock is represented by the process *Stop*. A process is deadlock-free, if it never reaches a state equivalent to *Stop*. On the semantical level, this intuition is captured as follows: A process $P$ is defined to be *deadlock-free* if and only if

$$\forall\, s \in \Sigma^* \bullet (s, \Sigma^{\checkmark}) \notin \mathit{failures}(P).$$

```
1  theory xq_refine_cpo = xq:
2
3  (* CPO apporach *)
4    defs FPmode_def [simp]: "FPmode == CPOmode"
5
6  (* Qa ⊑_T X *)
7    consts fx :: "ungPN => (gPN, Event) proc"
8    primrec "fx (X) = $Qa"
9
10   lemma x_refine_qa_cpo: "$Qa <=T ($X::(ungPN, Event) proc)"
11   apply (rule cspT_fp_induct_right[of _ _ "fx"], auto)
12   apply (induct_tac p, auto)
13   by (tactic {* cspT_unwind_tac 1 *})
14
15 (* X ⊑_T Qa *)
16   consts fa :: "gPN => (ungPN, Event) proc"
17   primrec "fa (Qa) = $X"
18
19   lemma qa_refine_x_cpo: "$X <=T ($Qa::(gPN, Event) proc)"
20   apply (rule cspT_fp_induct_right[of _ _ "fa"], auto)
21   apply (induct_tac p, auto)
22   by (tactic {* cspT_unwind_tac 1 *})
23
24 end
```

**Fig. 7.** A proof script for refinement over recursive process

After executing a trace $s$, the process $P$ is not allowed to refuse the set of possible communications $\Sigma^{\checkmark} = \Sigma \cup \{\checkmark\}$, namely those from the alphabet $\Sigma$ and the termination symbol $\checkmark$.

This definition on the semantical level has a purely syntactical counterpart: Roscoe [16] presents the theorem that $P$ is deadlock-free if and only if

$$DF^{\checkmark} \sqsubseteq_{\mathcal{F}} P,$$

where the process $DF^{\checkmark}$ is defined as $DF^{\checkmark} = (!\, x : \Sigma \rightarrow DF^{\checkmark}) \sqcap SKIP$ and $\sqsubseteq_{\mathcal{F}}$ stands for refinement in the stable failures model $\mathcal{F}$. As the stable failures model $\mathcal{F}$ has a complete axiomatic semantics [10], the above theorem yields that Csp has a complete calculus for checking deadlock-freedom. In Section 5.1 we demonstrate how to apply this calculus in Csp-Prover with an example taken from an industrial case study.

While the refinement approach completely abstracts from the structure of the concurrent system to be analyzed, other techniques have been suggested which are tailored specifically to a certain class of systems. Roscoe and Dathi [17], for instance, present the following theorem on a specific class of networks $V$: Let $V$ consist of a finite number of processes $\{P_1, \ldots, P_n\}$; if for all pairs of processes $P_i$ and $P_j$ with $i \neq j$ and for all states $\sigma$ of the network $V$ holds that

$$P_i \xrightarrow{\sigma}\bullet\ P_j \Rightarrow f_i(\sigma) > f_j(\sigma) \qquad (*)$$

then the network $V$ is deadlock free – see [17] for the full details of this theorem. The above formula reads: whenever the network is in a state $\sigma$ such that $P_i$ has an ungranted request to $P_j$ (written $P_i \xrightarrow{\sigma}\bullet\ P_j$), i.e., the system is in a

potential deadlock situation, then the progress $f_i(\sigma)$ which $P_i$ has made in state $\sigma$ is bigger than the progress $f_j(\sigma)$ which $P_j$ has made in state $\sigma$. Essentially, the theorem applies an argument on the exclusion of circular waiting to the network. The functions $f_k$, $1 \leq k \leq n$, are a so-called *variant* of the network $V$, which needs to be established for each network individually.

This technique is of purely semantical nature: the state to be considered in condition $(*)$ includes as one of its components the failures of the network to be analyzed. In this sense, the application of this theorem is more involved than the abstraction technique. Its advantage, however, lies in its capability to analyze whole families of systems. In [11] we give a proof of deadlock freedom for a whole family of systolic arrays. It is an open research question, if syntactic methods such as the refinement approach discussed above are capable of proving deadlock freedom for whole families of systems.

### 5.1 Proofs by abstraction

```
1  (* data part *)
2  typedecl init_d    typedecl request_d    typedecl response_d   typedecl exit_d
3  datatype Data = Init init_d | Exit exit_d | Request request_d | Response response_d
4  datatype Event = c Data
5
6  (* process part *)
7  datatype ACName = Terminal | TerminalConfigManagement
8                  | Acquirer | AcConfigManagement
9  consts ACfun :: "(ACName, Event) proc"
10 primrec
11   "ACfun (Terminal) = c !? init:(range Init) -> $TerminalConfigManagement"
12   "ACfun (TerminalConfigManagement) =
13      c ? x -> IF (x:range Request)
14               THEN c !? response:(range Response) -> $TerminalConfigManagement
15               ELSE IF (x:range Exit) THEN SKIP ELSE STOP"
16   "ACfun (Acquirer) = c ? x:(range Init) -> $AcConfigManagement"
17   "ACfun (AcConfigManagement) =
18      c !? exit:(range Exit) -> SKIP |~|
19      c !? request:(range Request)
20          -> c ? response:(range Response) -> $AcConfigManagement"
21 defs (overloaded) Set_ACfun_def [simp]: "PNfun == ACfun"
22
23 constdefs AC :: "(ACName, Event) proc"
24   "AC == ($Acquirer |[range c]| $Terminal)"
```

**Fig. 8.** EP2 Specification at the Abstract Component Description Level.

In the following, we give an example of how to prove deadlock-freedom in Csp-Prover via abstraction. To this end, we will show for a process `AC` that it is a refinement of a more abstract process `Abs` in the stable failures model $\mathcal{F}$. For this process `Abs` we prove $\texttt{\$DFtick} \sqsubseteq \texttt{Abs}$, where `$DFtick` is the Csp-Prover's representation of the process $DF^{\checkmark}$ defined above. As refinement is transitive, this establishes deadlock freedom for `AC`. It should be noted that Csp-Prover

```
1  datatype AbsName = Abstract | Loop
2  consts Absfun :: "(AbsName, Event) procDF"
3  primrec
4   "Absfun (Abstract) = c !? x -> $Loop"
5   "Absfun (Loop) =  c !? x -> (SKIP |~| c !? x -> $Loop)"
6  defs (overloaded) Set_Absfun_def [simp]: "PNfun == Absfun"
7
8  constdefs Abs :: "(AbsName, Event) proc"
9   "Abs == $Abstract"
```

**Fig. 9.** An abstraction of the process shown in Fig. 8.

also includes the proof of the above syntactical characterization of deadlock freedom.

Our example for this section is taken from an industrial case study on the verification of EP2 [1], which is a new standard of electronic payment systems. In [6], major parts of the EP2 system have been formalised in CSP-CASL [15]. For a system as complex as EP2, tool support is required in order to prove deadlock freedom for the interaction between the various components.

Translating the data part of the specifications given in [6] into adequate Isabelle code, we obtain specifications in the input format of CSP-Prover. Fig. 8 shows the nucleus[7] of the initialisation procedure of the EP2 `Terminal`. The `Terminal` starts the initialisation (line 11), where $\texttt{c!?x} : \texttt{X} \rightarrow \texttt{P}$ represents that one of data in $X$ is nondeterministically sent on `c`, and waits then for data sent by the `Acquirer`. If this data is of type `Request`, the `Terminal` answers with a value of type `Response` (line 14). Another possibility is that the `Acquirer` wants to exit the initialisation (line 15). Any other type of communication sent by the `Acquirer` will lead to a deadlock represented by the process `STOP` (line 15). On the other end of the communication, after receiving an initialisation request (line 16) the `Acquirer` internally decides if it wants to exit the process (line 18) or interact with the `Terminal` by sending a request followed by a response of the `Terminal` (line 19). The system `AC` to be analysed here consists of the parallel composition of the `Terminal` and the `Acquirer` synchronised on the channel `c` (line 24).

Using CSP-Prover, we can show in the stable-failure model $\mathcal{F}$ that the above described process `AC` is a refinement of the process `Abs` of Fig. 9. This proof gives another nice example on how to apply proof procedures to recursively defined processes in CSP-Prover. Note that `Abs` is a purely sequential process which can successfully terminate after any even number of communications through the channel `c`. Which specific data is sent on `c` can be ignored here.

Fig. 10 shows the complete script to prove the refinement relation $\texttt{Abs} \sqsubseteq_{\mathcal{F}} \texttt{AC}$ (line 15) in CSP-Prover. First, it is declared that the CMS approach is applied for the analysis of fixed points in this verification (line 2). Then we check that all processes are guarded. This check is fully automated routine (lines 5–6). Next,

[7] For the purpose of this paper, the specification text has been simplified. The complete formalisation and proof can be found in [8].

```
1  (* the CMS approach is used for analysing Fixed Points *)
2  defs FPmode_def [simp]: "FPmode == CMSmode"
3
4  (* check the guards *)
5  lemma guardedfun_AC_Abd[simp]: "guardedfun ACfun" "guardedfun Absfun"
6  by (simp add: guardedfun_def, rule allI, induct_tac p, simp_all)+
7
8  (* expected correspondence of process-names in Abs to AC *)
9  consts Abs_to_AC :: "AbsName => (ACName, Event) proc"
10 primrec
11   "Abs_to_AC (Abstract) = ($Acquirer |[range c]| $Terminal)"
12   "Abs_to_AC (Loop) = ($AcConfigManagement |[range c]| $TerminalConfigManagement)"
13
14 (* the main theorem *)
15 theorem ep2_Abs_AC: "Abs <=F AC"
16 apply (unfold Abs_def AC_def)
17 apply (rule cspF_fp_induct_left[of _ "Abs_to_AC"], auto)
18 apply (induct_tac p, auto)
19 apply ((tactic * cspF_hsf_tac 1 *)+,
20          (rule | rule cspF_decompo_ref |
21           rule cspF_Int_choice_left1, auto |
22           rule cspF_Int_choice_left2, rule cspF_decompo_ref, auto)?,
23          (auto simp add: image_iff inj_on_def)?)+
24 done
```

**Fig. 10.** The complete proof script for `Abs` $\sqsubseteq_{\mathcal{F}}$ `AC`.

a mapping is defined from the process-names of `Abs` to process expressions in `AC` (line 9-12)[8]. After these preparations, `Abs` $\sqsubseteq_{\mathcal{F}}$ `AC` is given as a goal (line 15). Using the above mapping, the proof obligations on the recursive process `Abs` are unfolded into a base case and step cases by `cspF_fp_induct_left`. This is the fixed point induction rule for unfolding the left-hand side (line 17). Since a step case is produced for each of the process names of `Abs`, the step cases are instantiated by induction on `AbsName` (line 18). Finally, the theorem is proven by Csp-Prover's tactic `csp_hsf_tac` (line 19) which sequentialises any expression, `csp_decompo_ref` which decomposes Csp-operators (line 20) by considering the refinement, for example,

$$Y \subseteq X \wedge (\forall\, x \in Y \,.\, P(s) \sqsubseteq_{\mathcal{F}} Q(s)) \implies !\, x : X \bullet P(s) \sqsubseteq_{\mathcal{F}} !\, x : Y \bullet Q(s),$$

`csp_Int_choice_left1` which selects the first $P$ from $P \sqcap Q$ in the left-hand side (line 21), and Isabelle's tactic `auto` (line 23).

It remains to show that `AC` is deadlock-free. Fig. 11 shows the complete script to prove the refinement relation \$`DFtick` $\sqsubseteq_{\mathcal{F}}$ `Abs` (line 8) in Csp-Prover. In this proof, `cspF_fp_induct_right`, which is the fixed point induction rule for unfolding the right-hand side, is applied (line 10). Since we use the CMS approach in this example, we can apply the fixed point induction to the both sides. Note that in the CPO approach `cspF_fp_induct_right` is available only for unfolding right-hand sides. The proof strategy used in Fig. 11 is similar to the case of `Abs` $\sqsubseteq$ `AC`.

[8] It is hard to automatically derive such correspondences. However, Csp-Prover can assist users to derive them.

```
1  (* expected correspondence of process-names in Abs to DF *)
2  consts Abs_to_DF :: "AbsName => (DFtickName, Event) proc"
3  primrec
4   "Abs_to_DF (Abstract) = ($DFtick)"
5   "Abs_to_DF (Loop) = ($DFtick)"
6
7  (* the main theorem *)
8  theorem ep2_DF_Abs: "$DFtick <=F Abs"
9  apply (unfold Abs_def)
10 apply (rule cspF_fp_induct_right[of _ _ "Abs_to_DF"], auto)
11 apply (induct_tac p, auto)
12 apply ((tactic {* cspF_hsf_tac 1 *}, rule cspF_Int_choice_left1,
13                                    rule cspF_decompo_ref, auto) |
14        (tactic {* cspF_hsf_tac 1 *}, rule cspF_Int_choice_left2, simp))+
15 done
```

**Fig. 11.** The complete proof script for $\texttt{\$DFtick} \sqsubseteq_{\mathcal{F}} \texttt{Abs}$.

```
1  theorem AC_isDeadlockFree: "AC isDeadlockFree"
2  apply (simp add: DeadlockFree_DFtick_ref)
3  apply (rule cspF_trans[of _ _ _ Abs])
4  apply (rule ep2_DF_Abs, rule ep2_Abs_AC)
5  done
```

**Fig. 12.** The complete proof script for the deadlock-freedom of AC.

Finally, we give a script for showing that AC is deadlock-free in Fig. 12. The main goal is rewritten to $\texttt{\$DFtick} \sqsubseteq \texttt{AC}$ by `DeadlockFree_DFtick_ref`, which is the encoded syntactical characterization of deadlock-freedom (line 2). We finally establish our proof goal by combining $\texttt{\$DFtick} \sqsubseteq \texttt{Aba}$ and $\texttt{Abs} \sqsubseteq \texttt{AC}$, which have been proved in Figs. 10 and 11, respectively.

## 6 Summary and Future work

In this paper, we have reviewed a number of proof principles of the process algebra Csp, have discussed their merits and limitations, and have shown by the means of practical examples how these techniques can be applied in the analysis of concurrent systems using Csp-Prover. It turned out that mechanized theorem proving is needed for both, the verification and the application of proof principles in Csp and process algebra in general.

Future work will include the implementation of further Csp models in Csp-Prover, the integration of FDR and Csp-Prover, as well as the application of Csp-Prover in further case studies.

*Acknowledgement* The authors would like to thank Liam O'Reilly and Temesghen Kahsai for help with some of the proof scripts, and Erwin R. Catesbeiana Jr for contributing deep insights into the nature of fixed point induction.

## References


1. *eft/pos 2000 Specification, version 1.0.1.* EP2 Consortium, 2002.
2. J. A. Bergstra, A. Ponse, and S. A. Smolka, editors. *Handbook of Process Algebra.* Elsevier Science Inc., New York, NY, USA, 2001.
3. B. Buth, J. Peleska, and H. Shi. Combining methods for the livelock analysis of a fault-tolerant system. In *AMAST'98*, LNCS 1548, pages 124–139. Springer, 1998.
4. B. Buth and M. Schrönen. Model-checking the architectural design of a fail-safe communication system for railway interlocking systems. In *FM'99*, LNCS 1709. Springer, 1999.
5. B. Dutertre and S. Schneider. Using a PVS embedding of CSP to verify authentication protocols. In *TPHOL 1997*, LNCS 1275, pages 121–136. Springer, 1997.
6. A. Gimblett, M. Roggenbach, and H. Schlingloff. Towards a formal specification of an electronic payment system in CSP-CASL. In *WADT 2004*, LNCS 3423, pages 61–78. Springer, 2005.
7. C. A. R. Hoare. *Communicating Sequential Processes.* Prentice Hall, 1985.
8. Y. Isobe and M. Roggenbach. Webpage on CSP-Prover. `http://staff.aist.go.jp/y-isobe/CSP-Prover/CSP-Prover.html`.
9. Y. Isobe and M. Roggenbach. A generic theorem prover of CSP refinement. In *TACAS 2005*, LNCS 3440. Springer, 2005.
10. Y. Isobe and M. Roggenbach. A complete axiomatic semantics for the CSP stable failures model. In *CONCUR 2006*, LNCS. Springer, 2006.
11. Y. Isobe, M. Roggenbach, and S. Gruner. Extending CSP-Prover by deadlock-analysis: Towards the verification of systolic arrays. In *FOSE 2005*, Japanese Lecture Notes Series 31. Kindai-kagaku-sha, 2005.
12. T. Kahsai, M. Roggenbach, and B.-H. Schlingloff. Specification-based testing for refinement. In *Proceedings of SEFM 2007.* IEEE Computer Society, 2007.
13. F. S. E. Limited. Failures-divergence refinement: FDR2. `http://www.fsel.com/`.
14. T. Nipkow, L. C. Paulon, and M. Wenzel. *Isabelle/HOL.* LNCS 2283. Springer, 2002.
15. M. Roggenbach. CSP-CASL: a new integration of process algebra and algebraic specification. *Theor. Comput. Sci.*, 354(1):42–71, 2006.
16. A. Roscoe. *The theory and practice of concurrency.* Prentice Hall, 1998.
17. A. W. Roscoe and N. Dathi. The pursuit of deadlock freedom. *Information and Computation*, 75(3):289–327, 1987.
18. B. Roscoe. Revivals, stuckness and responsiveness, 2005. Unpublished draft.
19. B. Roscoe. Seeing beyond divergence. In *Communicating Sequential Processes, the first 25 years*, LNCS 3525. Springer, 2005.
20. P. Ryan, S. Schneider, M. Goldsmith, G. Lowe, and B. Roscoe. *The Modelling and Analysis of Security Protocols: the CSP Approach.* Addison-Wesley, 2001.
21. V. Sassone, M. Nielsen, and G. Winskel. Models for concurrency. *Theoretical Computer Science*, 170(1–2):297–348, 1996.
22. S. Schneider. Verifying authentication protocol implementations. In *FMOODS 2002*, volume 209 of *IFIP Conference Proceedings*, pages 5–24. Kluwer, 2002.
23. H. Tej. *HOL-CSP: Mechanised Formal Development of Concurrent Processes.* BISS Monograph Vol. 19. Logos Verlag Berlin, 2003.
24. H. Tej and B. Wolff. A corrected failure-divergence model for CSP in Isabelle/HOL. In *FME'97*, LNCS 1313, pages 318–337. Springer, 1997.