Available online at www.sciencedirect.com

ScienceDirect

ELSEVIER

Electronic Notes in Theoretical Computer Science 250 (2009) 119–134

Electronic Notes in Theoretical Computer Science

www.elsevier.com/locate/entcs

The Stable Revivals Model in CSP-Prover

D. Gift Samuel[1] Markus Roggenbach[1]

Swansea University, United Kingdom

Yoshinao Isobe[1,2]

Information Technology Research Institute
National Institute of Advanced Industrial Science and Technology, Tsukuba, Japan

Abstract

The stable revivals model $\mathcal{R}$ provides a new semantic framework for the process algebra Csp. The model $\mathcal{R}$ has recently been added to the realm of established Csp models. Within the Csp context, it enhances the analysis of systems with regards to properties such as responsiveness and stuckness. These properties are essential in component based system design. In this paper we report on the implementation of different variants of the model $\mathcal{R}$ within Csp-Prover. Based on Isabelle/HOL, Csp-Prover is an interactive proof tool for Csp refinement, which is generic in the underlying Csp model. On the practical side, our encoding of the model $\mathcal{R}$ provides semi-automatic proof support for reasoning on responsiveness and stuckness. On the theoretical side, our implementation also yields a machine verification of the model $\mathcal{R}$'s soundness as well as of its expected properties.

Keywords: CSP, Isabelle/HOL, Stable Revivals Model.

1 Introduction

The process algebra Csp [6,15,20,1] 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, fault-tolerant systems and security protocols.

Fixing one syntax, Csp offers different semantic models, each of which is dedicated to special verification tasks. The traces model $\mathcal{T}$, e.g., covers *safety properties. Liveness 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 $\mathcal{N}$

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

[2] This work was supported by KAKENHI 20500023.

doi:10.1016/j.entcs.2009.08.021

allows for *livelock analysis*. The analysis of *fairness properties* requires models based on infinite traces, see [15,1] for further details.

Recently, the Csp realm of models has been extended by the newly designed *stable-revivals model* $\mathcal{R}$ [17]. On the practical side, the model $\mathcal{R}$ is appropriate to study *responsiveness* [18,19], which is a property significant in the context of component-based system design. From a theoretical point of view, the model $\mathcal{R}$ turns out to be fully abstract with respect to detecting when some system of processes can fail to make progress despite one or more of them having unfinished business with other(s). However, this comes at the price that certain algebraic properties fail to hold in the new model, among them the law $\sqcap - \Box$-distributivity $(P \Box Q) \sqcap R = (P \sqcap R) \Box (Q \sqcap R)$ is the most prominent example.

Csp-Prover [7,11,8,9,10] is an interactive theorem prover for Csp based on Isabelle/HOL [13]. With its theorem proving approach Csp-Prover complements the established model checker FDR [5] as a proof tool for Csp. Csp-Prover is generic in the various Csp models. Currently, it fully supports the traces model $\mathcal{T}$ and the stable-failures model $\mathcal{F}$.

Csp-Prover provides a deep encoding of Csp, and, consequently, also allows for meta theorems on the semantics to be implemented. Mistakes found, e.g., in the typing of one semantical function of the predecessor of the model $\mathcal{N}$ [23], or in the algebraic laws for the model $\mathcal{F}$ [8] demonstrates that presentations of similar models and axiom schemes will only be 'complete' once they have been accompanied by mechanised theorem proving [16].

In this paper, we report on the encoding the model $\mathcal{R}$ in Csp-Prover. Overall, our implementation validates the design as given in [17]. However, we suggest to adapt two semantic clauses and make a proposal of how to extend the model $\mathcal{R}$ – against the original intention to keep it as a model characterized by finite observations only – also to infinite alphabets.

In the remaining section, we briefly review related work. In [23], Tej and Wolff presented a Cpo based, shallow encoding of the failures divergence model $\mathcal{N}$ in Isabelle/HOL [13]. Camilleri mechanised the traces model $\mathcal{T}$ [2] and later a variation of the failures-divergence model $\mathcal{N}$ [3] in HOL. Dutertre and Schneider [4] formalised the traces model $\mathcal{T}$ in PVS [14] in order to reason about authentication protocols. Wei and Heather [24] extended this formalisation in order to reason about the stable failures model of Csp in PVS. Recently, Kammueller [12] formalised the Csp failures divergence model $\mathcal{N}$ in Isabelle/HOL. To the best of our knowledge, however, this is the first paper reporting on theorem proving for the model $\mathcal{R}$.

This paper is organised as follows: Section 2 introduces Csp-Prover, its syntax, and the underlying theorem prover Isabelle/HOL. Section 3 describes our implementation: we discuss three variants of the model $\mathcal{R}$ and study their properties w.r.t. completeness of the domains as well as type correctness and continuity of the operators. In Section 4, we report on our results concerning algebraic laws.

A status report on this project was published in [22], [21] includes the full technical details.

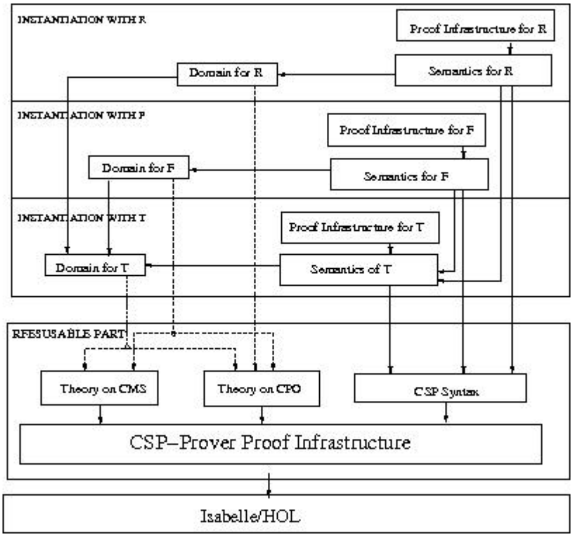

Fig. 1. Architecture of the model R in CSP-Prover

2 Background

In this section, first we give an overview of CSP-Prover, then summarise the syntax of the input language of CSP-Prover, CSP$_{\mathrm{TP}}$, and finally discuss how to work with the interactive theorem prover Isabelle/HOL.

2.1 CSP-*Prover*

CSP-Prover [7,11,8,9,10] is a proof tool for CSP based on the generic theorem prover Isabelle/HOL. CSP-Prover has a generic architecture, see Fig. 1. To this end, it includes a rich re-usable part, which is independent of the CSP model to be implemented. The re-usable part provides proof infrastructure such as standard operations and elementary theorems on traces and event sets; fixed point theories such as Tarski's fixed point theorem and the standard fixed point induction rule based on Complete Partial Orders (CPO) and Banach's fixed point theorem and the metric fixed point induction rule based on Complete Metric Spaces (CMS); and the CSP syntax. This re-usable part can then be instantiated with a specific CSP model. The instantiation of CSP-Prover with a CSP model, say the model $\mathcal{R}$, requires the following steps, see Fig. 1:

(i) to define the semantic domain (the box 'Domain for $\mathcal{R}$'),

(ii) to prove that this domain is a CPO (part of the dashed arrow from 'Domain for $\mathcal{R}$' to 'Theory on CPO'),

(iii) to define the semantic functions (the box 'Semantics for $\mathcal{R}$'),

(iv) to prove that these functions are type correct (the box 'Semantics for $\mathcal{R}$'),

(v) to prove that these functions are continuous (part of the dashed arrow from 'Domain for $\mathcal{R}$' to 'Theory on CPO'), and

(vi) to provide proof infrastructure for the model, i.e., prove algebraic laws and develop tactics specific to this model (the box ‘Proof Infrastructure for $\mathcal{R}$’).

2.2 *The input language of* CSP*-Prover*

CSP-Prover offers a rich set of CSP operators, where the language CSP$_{\mathrm{TP}}$ serves as semantical core, i.e., semantic clauses are given only for operators in this language. As usual, operators such as synchronous parallel, interleaving, sending and receiving values over channels are implemented as syntactic sugar.

Relatively to an arbitrary alphabet of communications Σ and a set of process-names Π, the language CSP$_{\mathrm{TP}}$, see Fig. 2, offers the usual set of CSP operators such as the basic processes *Skip*, *Stop*, and *Div*, action prefix, the binary choice operators, the call to a named process $\$N$ etc. CSP$_{\mathrm{TP}}$ extends the set of standard operators by

- a replicated internal choice operator $!!\, c : C \bullet P(c)$. This operator provides a ’tamed’ version of CSP’s general internal choice $\bigsqcap S$, which makes an internal choice over an arbitrary set of processes S. CSP$_{\mathrm{TP}}$ restricts this set S to be an indexed set of the form $\{P(c) \mid c \in C\}$ where the index set C is either $C \subseteq \mathbb{P}(\Sigma)$ or $C \subseteq \mathit{Nat}$ (i.e.$C \in \mathit{Choice}(\Sigma) = \mathbb{P}(\mathbb{P}(\Sigma)) \uplus \mathbb{P}(\mathit{Nat})$).
- a depth restriction operator $P \downarrow n$, where $P \downarrow n$ behaves exactly like P until exactly n events have occurred.

These two extensions allow one to express recursive process definitions by process terms only. Based on this insight, [8] presents a complete axiomatic semantics for CSP$_{\mathrm{TP}}$ w.r.t. the stable failures model $\mathcal{F}$. We use $\mathit{Proc}_{(\Pi,\Sigma)}$ for representing the set of processes defined by Fig. 2.

$$
\begin{array}{rll}
P ::= & \mathit{Skip} & \text{\%\% successful terminating process} \\
\mid & \mathit{Stop} & \text{\%\% deadlock process} \\
\mid & \mathit{Div} & \text{\%\% divergence} \\
\mid & a \rightarrow P & \text{\%\% action prefix} \\
\mid & ?\, x : X \rightarrow P(x) & \text{\%\% prefix choice} \\
\mid & P \,\Box\, P & \text{\%\% external choice} \\
\mid & P \sqcap P & \text{\%\% internal choice} \\
\mid & !!\, c : C \bullet P(c) & \text{\%\% replicated internal choice} \\
\mid & \mathit{if}\ b\ \mathit{then}\ P\ \mathit{else}\ P & \text{\%\% conditional} \\
\mid & P \,[\![X]\!]\, P & \text{\%\% generalised parallel} \\
\mid & P \setminus X & \text{\%\% hiding} \\
\mid & P[\![r]\!] & \text{\%\% relational renaming} \\
\mid & P \,\mathbin{;}\, P & \text{\%\% sequential composition} \\
\mid & P \downarrow n & \text{\%\% depth restriction} \\
\mid & \$N & \text{\%\% process name}
\end{array}
$$

$X \subseteq \Sigma$, $C \in \mathit{Choice}(\Sigma)$, b is a condition, $r \subseteq (\Sigma \times \Sigma)$, $n \in \mathit{Nat}$, and $N \in \Pi$.

Fig. 2. Syntax of basic CSP processes in CSP-Prover.

For the purpose of implementing the stable revivals model $\mathcal{R}$, we extend our core language Csp$_{\mathrm{TP}}$ by the Csp interrupt operator $P \triangle Q$ and the Csp timeout operator $P \triangleright Q$ as both these operators play prominent roles within the model $\mathcal{R}$.

2.3 Isabelle/HOL

Isabelle/HOL [13] is a widely used, interactive theorem prover for Higher Order Logic. It provides a large range of built-in data types like `bool`, `int`, `nat`, etc. More complex types are provided via type constructors such as `list`, the type of lists, `set`, the type of sets, or `pair`, encoding the Cartesian product. This given type system can be extended in various ways, e.g., by a new `datatype`. Csp-Prover uses this construct, for example, in order to create the polymorphic type `’a event`:

```
datatype ’a event = Ev ’a | Tick
```

Here, `’a` is a type variable. Assuming that `’a` represents the alphabet of communications Σ, the new type `’a event` represents its extension $\Sigma^{\checkmark}$ by the termination signal $\checkmark$.

Type definitions are yet another technique to extend Isabelle/HOL’s type system. They define a sub-type of an existing type. In Csp-Prover this is used, e.g., in order to define the traces over an alphabet:

```
typedef ’a trace = "{l::(’a event list).  Tick ~:  set (butlast l)}
```

The above code creates the type `’a trace` as a subtype `’a event list`: it includes all lists over events, where the event tick does not appear but in the last position – `butlast l` removes the last element from a list, the function `set` forms the set of all elements appearing in a list.

New function symbols are declared using the keyword `consts`. Their meaning can then be defined, e.g., using the keyword `defs`, which declares them to be a definitional extension of the existing theory. The command `constdef` combines these two mechanisms, which in Csp-Prover is, e.g., used in order to encode Csp healthiness conditions:

```
constdef
 HC_T1 ::  "’a trace set => bool"
 HC_T1_def :  "HC_T1 T == (T ~= {} & prefix_closed T)"
```

The above code declares a check function `HC_T1` for the traces condition **T1**. The function `HC_T1` returns ‘true’ if its parameter `T` is non-empty and prefix closed.

Primitive recursion is another mechanism to define the meaning of a function in Isabelle/HOL. This mechanism requires the domain of the function to be defined with the Isabelle/HOL `datatype` construct. Given Csp-Prover’s type for processes:

```
datatype
 (’p,’a) proc
 = STOP  | SKIP  | DIV | ...
```

which takes a type `’p` for process names and a type `’a` for the alphabet of communications as its parameters, the semantic functions of the traces model $\mathcal{T}$ are defined inductively over the process syntax:

```
const
 traces ::  "('p,'a) proc => ('p => 'a domT) => 'a domT"
primrec
 "traces(STOP) = (%M. {<>}t)"
 "traces(SKIP) = (%M. {<>, <Tick>}t)" ...
```

Here, the first parameter of the function `traces` is the basic Csp process formed along the extended grammar given in Fig. 2, the second parameter gives the interpretation of the process names. In the semantic clauses, this second parameter appears as the lambda abstraction `%M`. Note that both clauses presented above are independent of the process interpretation. The function `{_}t` maps the given sets into the traces domain.

Types, their operations, and theorems can be grouped into axiomatic type classes. Csp-Prover uses this abstraction mechanism e.g. in order to collect the standard results on complete partial orders. Instantiation makes these results then available for a specific settings:

```
instance domT ::  (type) cpo
 apply (intro_classes)
 apply (simp add:  hasLUB_def)
 apply (rule_tac x="UnionT X" in exI)
 apply (simp add:  directed_def UnionT_isLUB)
 done
```

In the code above, Csp-Prover's type class `cpo` is instantiated with the domain `domT` for the Csp traces model. This instantiation generates proof obligations, which essentially are discharged by referring to the theorem `UnionT_isLUB` which states that the set theoretic union provides the least upper bound in the traces domain.

3 Implementation

The model $\mathcal{R}$ appeared first in [19]. The main source for our paper is [17], which is the revision of a 2005 draft. In this section, we give variations of the stable revivals model, encode them in Csp-Prover, and discuss type correctness and continuity of semantic operators.

3.1 The domain of the model

Given a set Σ of communications, the stable revivals model denotes a Csp process P by a triple (T, D, R), where

- $T \subseteq \Sigma^{*\checkmark}$ consists of all P's finite traces.
- $D \subseteq \Sigma^{*}$ consists of all traces after which P can possibly deadlock. Since a successfully terminated trace (a trace ending with $\checkmark$) can not lead to a deadlock, $\checkmark$ does not appear in the deadlock traces.
- $R \subseteq \Sigma^{*} \times \mathcal{P}(\ \Sigma) \times\ \Sigma$ consists of all revivals of P. A revival is a triple in which the first element is a trace of the process P, the second element is the refusal set in

a stable state after the given trace and the third element is a non-tick event that the process P can accept in the stable state after the trace. The third element is called "reviving event". In a stable state of revival, the process does not engage in ✓ or any internal events. A revival (s, X, a) states that process P can perform trace s, stably refuse X, and accept the event a.

As usual in Csp, not all such triples are included in the semantical domain of the model. Only a 'healthy' subset is selected. Given a triple (T, D, R) over an alphabet Σ, [17] lists the following healthiness conditions:

T1 T is nonempty and prefix-closed.

D1 $D \subseteq T$.

R1 $(s, X, a) \in R \Rightarrow s \frown \langle a \rangle \in T$, i.e. every trace implied by a revival is required to be in T.

R2 $(s, X, a) \in R \wedge Y \subseteq X \Rightarrow (s, Y, a) \in R$, i.e. if there is a revival that can stably refuse X, then all subsets Y of X can also be refused.

R3 $(s, X, a) \in R \wedge b \in \Sigma \Rightarrow ((s, X, b) \in R \vee (s, X \cup \{b\}, a) \in R)$, i.e. if a state has a revival, then every event b of the alphabet must appear in the refusal set of a revival or must be an accepted event.

RRS $(s, X, a) \in R \Rightarrow a \notin X$, i.e. the accepted event a is not allowed to appear in the refusal set X.

We also consider the condition:

R3′ $((s, X, a) \in R \wedge Y \subseteq \Sigma \wedge \forall\, b \in Y : (s, X, b) \notin R) \Rightarrow (s, X \cup Y, a) \in R$, i.e. any set of events that is not accepted when X is refused must also be refused. This condition results from our attempts to come up with a type correct model over arbitrary alphabets.

Varying the included healthiness conditions, we define three different domains $domR_{\Sigma}^{fin}$, $domR_{\Sigma}^{arb}$, and $domR_{\Sigma}^{m}$ as subsets of

$$\Sigma^{*\checkmark} \times \Sigma^{*} \times (\Sigma^{*} \times \mathcal{P}(\Sigma) \times \Sigma)$$

The triples (T, D, R) of

$domR_{\Sigma}^{fin}$ satisfy the healthiness conditions **T1, D1, R1, R2, R3** and **RRS**. Furthermore, Σ is required to be finite. This is the definition of the domain as given in [17].

$domR_{\Sigma}^{arb}$ satisfy the healthiness conditions **T1, D1, R1, R2, R3** and **RRS**. Σ, however can be an arbitrary set.

$domR_{\Sigma}^{m}$ satisfy the healthiness conditions **T1, D1, R1, R2, R3′** and **RRS**. We call $domR_{\Sigma}^{m}$ the modified domain.

In general, these domains relate as follows:

$$domR_{\Sigma}^{arb} \supset domR_{\Sigma}^{m}.$$

As **R3′** implies **R3**, $domR^{m}_{\Sigma}$ is a subset of $domR^{arb}_{\Sigma}$. There exists alphabets, for which this set inclusions is proper. Let, e.g., $\Sigma = Nat \cup \{a, b\}$. Then

$$D_{ex} = (D_T, D_D, D_R) = (\{\langle\rangle, \langle a\rangle, \langle b\rangle\},\ \{\},\ \{(\langle\rangle, X, a),\ (\langle\rangle, X, b) \mid X \subseteq_{fin} Nat\})$$

is an element of $domR^{arb}_{\Sigma}$ but not of $domR^{m}_{\Sigma}$. Clearly, **R3** holds for $a, b \in \Sigma$. Let $n \in Nat$. Then $(\langle\rangle, \{n\}, a) \in R_D$, as $\{n\}$ is a finite set. Concerning **R3′**, instantiate in its premise $(s, X, a) = (\langle\rangle, \emptyset, a) \in D_R$ and $Y = Nat$. Then $\forall\, n \in Nat : (\langle\rangle, X, n) \notin D_R$. However, $(\langle\rangle, \emptyset \cup Nat, a) \notin D_R$. Thus, $D_{ex} \notin domR^{m}_{\Sigma}$.

We strengthen condition **R3** to **R3′** in order to obtain a type-correct semantics for the renaming operator over arbitrary alphabets. Presumably, including **R3′** also leads to a *surjective* semantic mapping from the set of processes $Proc_{(\Pi,\Sigma)}$ to the domain $domR^{m}_{\Sigma}$, i.e. the model $\mathcal{R}$ with *infinite* alphabet Σ is expressive. [17] shows in Theorem 5.3 that there is a process for each member of $domR^{fin}_{\Sigma}$ with finite alphabet. However, if infinite alphabets are allowed, $domR^{arb}_{\Sigma}$ includes *meaningless* members to which no process is mapped, where the above example $D_{ex} = (D_T, D_D, D_R)$ is such a member of $domR^{arb}_{\Sigma}$.

In accordance to [17], in all three domains we take componentwise subset inclusion as the ordering relation.

3.2 Encoding the domains of the model $\mathcal{R}$

In this subsection, we prove that all three domains presented above are pointed Cpos. To this end, we prove that the individual components of the models are pointed Cpos. Technically, this is achieved by proving the models to be instances of axiomatic classes defined in Csp-Prover. We demonstrate our technique on selected examples.

First, we create a new type to represent the domain of the model. The domain of the model $\mathcal{R}$ is encoded using the domain of the model $\mathcal{T}$. We create types to represent the deadlock and revivals component of the model separately. The command `types` creates a type synonym called `’a revival` to store a trace, a set of events and an event. The type of the revivals component is created by the following command in Isabelle/HOL:

```
types  ’a revival = "(’a trace * ’a event set * ’a event) "
typedef ’a setR = "{ X ::(’a revivals set).
                     HC_RT(R) & HC_RF(R) & HC_R2(R) & HC_R3(R) }"
```

The power-set of the domain of the revivals component without healthiness conditions is represented as the type `’a revivals set`. We restrict the type `’a revivals set` with healthiness conditions relevant for the revivals component only. `HC_R2(R)` and `HC_R3(R)` encode healthiness conditions **R2** and **R3**, respectively, introduced in Section 3.1, and `HC_RT(R)` and `HC_RF(R)` require that the s and X of a revival (s, X, a) do not contain $\checkmark$ and that a is different from $\checkmark$.

In order to turn `setR` into a partially ordered set, we declare the type `setR` to be an instance of the axiomatic type classes `ord` and `order`:

```
instance setR ::  (type) ord
 by (intro_classes)
 defs (overloaded)
 subsetR_def :  "F <= E == Rep_setR (F) <= Rep_setR (E)"
 psubsetR_def :  "F < E == Rep_setR (F) < Rep_setR (E)"
 instance setR ::  (type) order
```

We prove that `setR` is closed under union as stated in the following lemma:

```
 lemma setR_Union_in_setR: "(Union (Rep_setR ‘ Fs)) :  setR"
```

This lemma is then used in proving `setR` to be a Cpo.

Similarly we can create a type `'a setD` to represent the deadlock component, and prove that it is a pointed Cpo.

In the next step, we implement the domain `Domain_R` based on the theories `Domain_T`, `set_R`, and `set_D` for the individual components of the model $\mathcal{R}$. The type of domain of the model $\mathcal{R}$, `'a domR`, is created using the type of the traces, deadlocks and revivals component. The remaining healthiness conditions, which connect the different components, are encoded in the type definition of the domain:

```
types 'a domTsetDsetR = "('a domT * 'a setD * 'a setR)"
typedef 'a domR = "{ X ::('a domTsetDsetR). HC_D1(X) & HC_R1(X) }"
```

Here, `HC_D1(R)` and `HC_R1(R)` represent the healthiness conditions **D1** and **R1**, respectively. Using a lemma from the reusable part of Csp-Prover, we can lift the property ‘pointed cpo’ from the components `domT`, `setD`, and `setR` to their product `domTsetDsetR`. Next, we show that least upper bounds preserve the healthiness conditions **D1** and **R1**. Finally, we prove that the product space of `domR` is also a pointed Cpo, again with a lemma given in the reusable part of Csp-Prover.

Using this approach, we have verified in Isabelle/HOL that the domains $domR_{\Sigma}^{arb}$ and $domR_{\Sigma}^{m}$ form pointed cpos. As the type $domR_{\Sigma}^{arb}$ is a super type of $domR_{\Sigma}^{fin}$, on the meta level we can conclude that also $domR_{\Sigma}^{fin}$ is a pointed cpo.

3.3 Semantic clauses

In this section we define two sets of semantic clauses for the stable revivals model: the set Sem^{orig} follows [17]; relatively to this, the set Sem changes two deadlock-clauses in order to obtain Csp standard laws – see Section 4 for the details.

Process denotations in the stable revivals model are given with the help of three component functions, namely *traces*, *deadlocks*, and *revivals*. As discussed in Section 2.3, Csp-Prover defines the semantics of the Csp operators relatively to an interpretation M of the process names. The semantic clauses Sem^{orig} are literally taken from [17], however extended to our input language as follows:

- For the replicated internal choice we adapt the clauses given for general internal choice [3] :

[3] [17] only defines the revivals component of the general internal choice operator. In the traces component, for simplicity we follow the – admittedly questionable – Csp-Prover tradition to add the set $\{\langle\rangle\}$ which

$$\begin{aligned} \mathit{traces}_M(!!\ c : C \bullet P(c)) &= \bigcup\{\mathit{traces}_M(P(c)) \mid c \in C\} \cup \{\langle\rangle\} \\ \mathit{deadlocks}_M(!!\ c : C \bullet P(c)) &= \bigcup\{\mathit{deadlocks}_M(P(c)) \mid c \in C\} \\ \mathit{revivals}_M(!!\ c : C \bullet P(c)) &= \bigcup\{\mathit{revivals}_M(P(c)) \mid c \in C\} \end{aligned}$$

- For the depth restriction operator we define [4]:

$$\begin{aligned} \mathit{traces}_M(P \downarrow n) &= \{s \in \mathit{traces}_M(P) \mid \sharp s \leq n\} \\ \mathit{deadlocks}_M(P \downarrow n) &= \{s \in \mathit{deadlocks}_M(P) \mid \sharp s < n\} \\ \mathit{revivals}_M(P \downarrow n) &= \{(s, X, a) \in \mathit{revivals}_M(P) \mid \sharp s < n\} \end{aligned}$$

The set Sem makes the following changes in the semantic clauses:

- For the prefix choice operator, the original deadlock component is:
  $\mathit{deadlocks}_M^{orig}(?\ x : A \rightarrow P(x)) = \{\langle x\rangle ^\frown t' \mid t' \in \mathit{deadlocks}_M^{orig}(P(x)), x \in A\}$
  Here we add the empty trace for the case that the choice set A is empty [5]:
  $$\begin{aligned} \mathit{deadlocks}_M(?\ x : A \rightarrow P(x)) = &\ \{\langle x\rangle ^\frown t' \mid t' \in \mathit{deadlocks}_M(P(x)), x \in A\} \\ \cup &\ \{\langle\rangle \mid A = \{\}\} \end{aligned}$$
  I.e. prefix choice over the empty trace yields an immediate deadlock, for example,
  $$\mathit{deadlocks}_M^{orig}(?\ x : \emptyset \rightarrow \mathit{Skip}) = \emptyset,$$
  $$\mathit{deadlocks}_M(?\ x : \emptyset \rightarrow \mathit{Skip}) = \{\langle\rangle\}$$

- For relational renaming, the original deadlock component is:
  $\mathit{deadlock}_M^{orig}(P[[R]]) = \{ s' \mid \exists\, s\,.\, sR^* s' \wedge s \in \mathit{deadlocks}_M^{orig}(P)\}$
  Here we take instead all those traces s', whose origin s has a failure that leads to a deadlock after renaming:
  $\mathit{deadlock}_M(P[[R]]) = \{ s' \mid \exists\, s\,.\, sR^* s' \wedge (s, R^{-1}(\Sigma^{\checkmark})) \in \mathit{failures}_M(P)\}$
  [17] defines this function $\mathit{failures}_M(P)$ with help of $\mathit{traces}_M(P)$, $\mathit{failures}_M(P)$, and $\mathit{revivals}_M(P)$ as follows:
  $$\begin{aligned} \mathit{failures}_M(P) = &\ \{(s, X) \mid X \subseteq \Sigma^{\checkmark} \wedge s \in \mathit{deadlocks}_M(P)\} \\ \cup &\ \{(s, X), (s, X \cup \{\checkmark\}) \mid (s, X, a) \in \mathit{revivals}_M(P)\} \\ \cup &\ \{(s, X) \mid s ^\frown \langle\checkmark\rangle \in \mathit{traces}_M(P) \wedge X \subseteq \Sigma\} \\ \cup &\ \{(s ^\frown \langle\checkmark\rangle, X) \mid s ^\frown \langle\checkmark\rangle \in \mathit{traces}_M(P) \wedge X \subseteq \Sigma^{\checkmark}\} \end{aligned}$$
  The modified clause $\mathit{deadlock}_M(P[[R]])$ has the effect that the renaming relation can lead to deadlock traces, namely in the case that the domain of renaming relations is a proper subset of the alphabet [6]. Take for example the alphabet $\Sigma = \{a\}$ with the empty renaming relation $\emptyset$. The process $a \rightarrow \mathit{Skip}$ does not lead to any deadlock, consequently, after renaming the original deadlock clause does not yield a trace leading to a deadlock:
  $$\mathit{deadlocks}_M^{orig}((a \rightarrow \mathit{Skip})[[\emptyset]]) = \{s' \mid \exists\, s.\, s\emptyset^* s' \wedge s \in \emptyset\} = \emptyset$$
  In our example, $R^{-1}(\Sigma^{\checkmark}) = \emptyset^{-1}(\{a, \checkmark\}) = \{\checkmark\}$. Consequently, our modified

extends the operator also to the empty choice set. Our definition of the deadlock component to be the union of all individual deadlock components follows the line of thought in the clause for binary internal choice, which [17] defines as $\mathit{deadlocks}_M(P \sqcap Q) = \mathit{deadlocks}_M(P) \cup \mathit{deadlocks}_M(Q)$.

[4] The traces component is as given in [15].

[5] This change has been approved by B. Roscoe.

[6] Some authors assume for Csp the static condition that the domain of a renaming relation R has to cover the full process alphabet of P – we give here a semantics that does not rely on this condition.

clause yields the result:

$$\mathit{deadlocks}_M((a \to \mathit{Skip})[[\emptyset]]) = \{s' \mid \exists\, s.\, s\emptyset^* s' \wedge (s, \{\checkmark\}) \in \{(\langle\rangle, \{\checkmark\}), \ldots\}\} = \{\langle\rangle\}$$

With these sets of semantic clauses and given an interpretation of process names M, the meaning of each process $P \in \mathit{Proc}_{(\Pi,\Sigma)}$ is defined by

$$[\![P]\!]_{\mathcal{R}(M)} = (\mathit{traces}_M(P), \mathit{deadlock}_M(P), \mathit{revivals}_M(P)).$$

3.4 Implementation of the semantic clauses and proof of type correctness

The Isabelle/HOL encoding of the semantic clauses appears to be straight forward. One would like to define the component functions *traces*, *deadlocks*, and *revivals* separately, where the traces component shall be inherited from the implementation of the traces model $\mathcal{T}$ in Csp-Prover. However, this is not possible, as the semantic clause of the generalized parallel operator makes the *deadlock* clauses and the *revivals* clause mutually dependent.

Thus, we define in a first phase deadlocks and revivals simultaneously as one function:

```
DeadlockRevivals ::  "('p,'a) proc => ('p => 'a domR) => ('a setD * 'a setR)"
```

From this function we separate in a second phase the individual definition of the revivals and deadlocks:

```
consts
 deadlocks ::  "('p,'a) proc => ('p => 'a domR) => 'a setD"
 revivals ::  "('p,'a) proc => ('p => 'a domR) => 'a setR"
defs
 deadlocks_def:  "deadlocks (P) M == ( (fst ( (DeadlockRevivals(P) M ))))"
 revivals_def:  "revivals (P) M == ( (snd ( (DeadlockRevivals(P) M ))))"
```

and establish the original semantic clauses as lemmas. Overall, this approach makes proofs easier and keeps the encoding readable.

Given an interpretation M of the process names, the range of the semantic function $[\![\cdot]\!]_{\mathcal{R}(M)}$ given in the previous section is clearly a subset of $\Sigma^{*\checkmark} \times \Sigma^* \times (\Sigma^* \times \mathcal{P}(\Sigma) \times \Sigma)$. It is, however, unclear, whether the semantic clauses are *type correct*, i.e., whether they produce healthy elements only provided their arguments are healthy. The modified domain presented in Section 3.1 is motivated by the fact that renaming fails to be type correct in the case of an infinite alphabet.

We proved the type correctness of renaming provided that the renaming relation R is finite over the domains $\mathit{domR}^{arb}_{\Sigma}$ in both version of semantic clauses. However, type correctness fails over the domains $\mathit{domR}^{arb}_{\Sigma}$ when the relation R is allowed to be infinite. The cause for this problem lies in the revival component, which is determined as follows:

$$\begin{aligned}\mathit{revivals}_M(P[[R]]) = \{(s', X, a') \mid \exists\, s, a\,.\, sR^*s' \wedge\ a\,R\,a' \wedge \\ (s, R^{-1}(X), a) \in \mathit{revivals}_M(P)\}\end{aligned}$$

Let now $\Sigma = \mathit{Nat} \cup \{a, b\}$, let N be a process name, let $M(N) = D_{ex}$ denote the element D_{ex} from Section 3.1, let $\mathit{Rel} \subseteq \Sigma \times \Sigma$ be the renaming relation with $\mathit{Rel} = \{(a, a)\} \cup \{(n, b) \mid n \in \mathit{Nat}\}$, i.e., a is renamed into a, all natural numbers n are renamed into b, and b is not in the domain of Rel. Then $D' = \mathit{revivals}(\$N[[Rel]]) = \{(\langle\rangle, \emptyset, a)\}$. Healthiness condition **R3** does not hold for D': as $(\langle\rangle, \emptyset, a) \in D'$ and $b \in \Sigma$ we need to have $(\langle\rangle, \emptyset, b) \in D'$ or $(\langle\rangle, \{b\}, a) \in D'$, which both is not the case. Condition $\mathbf{R3'}$ eliminates elements like D_{ex} from the domain. Using Isabelle/HOL, we could established: over the domain $domR^m_\Sigma$ renaming is type correct.

Summarising we can state: using Isabelle/HOL we have established that, provided renaming is finite, the two sets of semantical clauses Sem^{orig} and Sem are type correct over $domR^{arb}_\Sigma$. On the meta level we can conclude from this that they are also type correct over $domR^{fin}_\Sigma$. We have also proven using Isabelle/HOL that over $domR^m_\Sigma$ both sets of clauses Sem^{orig} and Sem are type correct without any restrictions.

3.5 Recursive Processes and continuity

Csp$_{\mathtt{TP}}$ provides a special function $\mathtt{PNfun}_\Pi : \Pi \rightarrow Proc_{(\Pi,\Sigma)}$, which is called a *process-name function*, in order to describe recursive equations:

$$\begin{array}{lcl} \mathtt{PNfun}_\Pi \ A & = & a \rightarrow \$B \\ \mathtt{PNfun}_\Pi \ B & = & (b \rightarrow \$A) \,\Box\, (c \rightarrow Skip) \end{array}$$

Here, the set of Π can be infinite, i.e., also infinite state processes can be expressed.

The interpretation M has to satisfy the equation[7]: for all $N \in \Pi$,

$$[\![\$N]\!]_{\mathcal{R}(M)} = [\![\mathtt{PNfun}_\Pi(N)]\!]_{\mathcal{R}(M)}$$

Since $[\![\$N]\!]_{\mathcal{R}(M)} = M(N)$, this can be rewritten to the following form:

$$M = [\![\mathtt{PNfun}_\Pi]\!]^{fun}_{\mathcal{R}}(M)$$

where $[\![\mathtt{PNfun}_\Pi]\!]^{fun}_{\mathcal{R}}(M) = (\lambda\, N.\ [\![\mathtt{PNfun}_\Pi(N)]\!]_{\mathcal{R}(M)})$. Consequently, M is a fixed point of the function $[\![\mathtt{PNfun}_\Pi]\!]^{fun}_{\mathcal{R}}$.

We have encoded the cpo approach for the stable revivals model $\mathcal{R}$, where the ideal interpretation, written $\mathtt{MR}_\Pi$, is given as follows:

$$\mathtt{MR}_\Pi = \mathtt{LFP}([\![\mathtt{PNfun}_\Pi]\!]^{fun}_{\mathcal{R}})$$

where $\mathtt{LFP}$ represents the least fixed point. Finally, the semantics $[\![P]\!]_{\mathcal{R}}$ of each process P is defined as follows: $[\![P]\!]_{\mathcal{R}} = [\![P]\!]_{\mathcal{R}(\mathtt{MR}_\Pi)}$. Consequently,

$$[\![\$N]\!]_{\mathcal{R}} = [\![\$N]\!]_{\mathcal{R}(\mathtt{MR}_\Pi)} = [\![\mathtt{PNfun}_\Pi(N)]\!]_{\mathcal{R}(\mathtt{MR}_\Pi)} = [\![\mathtt{PNfun}_\Pi(N)]\!]_{\mathcal{R}}.$$

In the rest of this section, we prove that the least fixed point of $[\![\mathtt{PNfun}_\Pi]\!]^{fun}_{\mathcal{R}}$ always exists for any process-name function $\mathtt{PNfun}_\Pi$ using Tarski's fixed point theorem. In order to apply the Tarski's theorem, we have to prove that the product space of domain $\Pi \rightarrow domR^m_\Sigma$ is Cpo and the function $[\![\mathtt{PNfun}_\Pi]\!]^{fun}_{\mathcal{R}}$ is continuous. We have already discussed the Cpo property in Section 3.2. Thus, we explain the continuity here.

[7] The semantics of P with the interpretation $M : \Pi \rightarrow domR^m_\Sigma$ is defined by

$$[\![P]\!]_{\mathcal{R}(M)} = (traces_{(fst(M))}(P), deadlock_M(P), revivals_M(P)).$$

The reusable part of CSP-Prover has the following lemma to prove that every component function is continuous if and only if the composition is continuous:

$$[\![f]\!]^{fun}_{\mathcal{R}} \text{ is continuous } \Leftrightarrow\ \forall\, p \in \Pi.\ (\lambda\, M.\ [\![f(p)]\!]_{\mathcal{R}(M)}) \text{ is continuous.}$$

Therefore, at first, we show that the component function $(\lambda\, M.\, [\![P]\!]_{\mathcal{R}(M)})$ is continuous for each process $P \in \mathit{Proc}_{(\Pi,\Sigma)}$.

Lemma 3.1 *Let $P \in \mathit{Proc}_{(\Pi,\Sigma)}$. Then, $(\lambda\, M.\, [\![P]\!]_{\mathcal{R}(M)}) : (\Pi \to \mathit{domR}^m_\Sigma) \to \mathit{domR}^m_\Sigma$ is continuous.*

Proof. We use a fact that the pair function $(\lambda\, x.\, (f(x), g(x)))$ is continuous if their components f, g are continuous. Thus, this lemma can be proven by showing that all $(\lambda\, M.\, \mathit{traces}_{(\mathit{fst}(M))}(P))$, $(\lambda\, M.\, \mathit{deadlock}_M(P))$, and $(\lambda\, M.\, \mathit{revivals}_M(P))$ are continuous. Here, we only show the continuity of $(\lambda\, M.\, \mathit{revivals}_M(P))$, in other words, we show that $\bigsqcup\{\mathit{revivals}_M(P) \mid M \in \Delta\} = \mathit{revivals}_{(\bigsqcup \Delta)}(P)$ by structural induction on P, for each directed set Δ.

- The case of $P = b \to P'$. For showing the subset relation ($\subseteq$), let $(s, X, a) \in \bigsqcup\{\mathit{revivals}_M(P) \mid M \in \Delta\}$, thus for some $M \in \Delta$,

  $$\begin{aligned}(s, X, a) &\in \mathit{revivals}_M(b \to P') \\ &= \{(\langle\rangle, X, b) \mid b \notin X\} \cup \{(\langle b\rangle \frown s', X, a) \mid (s', X, a) \in \mathit{revivals}_M(P')\}\end{aligned}$$

  Therefore, the following two cases are possible.
  · The case of $s = \langle\rangle$ and $a = b \notin X$. Hence,

  $$(s, X, a) \in \{(\langle\rangle, X, b) \mid b \notin X\} \subseteq \mathit{revivals}_{(\bigsqcup \Delta)}(b \to P')$$

  · The case of $s = \langle b\rangle \frown s'$ and $(s', X, a) \in \mathit{revivals}_M(P')$ for some s'. It implies $(s', X, a) \in \bigsqcup\{\mathit{revivals}_M(P') \mid M \in \Delta\}$. By the induction hypothesis, $(s', X, a) \in \mathit{revivals}_{(\bigsqcup \Delta)}(P')$. Hence,

  $$\begin{aligned}(s, X, a) &\in \{(\langle b\rangle \frown s', X, a) \mid (s', X, a) \in \mathit{revivals}_{(\bigsqcup \Delta)}(P')\} \\ &\subseteq \mathit{revivals}_{(\bigsqcup \Delta)}(b \to P')\end{aligned}$$

  Consequently, $\bigsqcup\{\mathit{revivals}_M(P) \mid M \in \Delta\} \subseteq \mathit{revivals}_{(\bigsqcup \Delta)}(P)$.
  The reverse relation ($\supseteq$) can be proven by a symmetric argument.

- The case of $P = \$N$ is proven as follows: Because the product space of the stable revival domain $(\Pi \to \mathit{domR}^m_\Sigma)$ is complete and Δ is directed, $\bigsqcup \Delta$ exists. Furthermore, we can prove $(\bigsqcup \Delta)(N) = \bigsqcup\{M(N) \mid M \in \Delta\}$. Hence,

  $$\begin{aligned}\bigsqcup\{\mathit{revivals}_M(\$N) \mid M \in \Delta\} &= \bigsqcup\{M(N) \mid M \in \Delta\} = (\bigsqcup \Delta)(N) \\ &= \mathit{revivals}_{(\bigsqcup \Delta)}(\$N)\end{aligned}$$

- The other cases can be proven similarly.

□

This result is easily extended over the product space of the domain domR^m_Σ by the lemma given in the reusable part of CSP-Prover.

Lemma 3.2 *Let $f \in \Pi \to \mathit{Proc}_{(\Pi,\Sigma)}$. Then, $[\![f]\!]^{fun}_{\mathcal{R}} : ((\Pi \to \mathit{domR}^m_\Sigma) \to (\Pi \to \mathit{domR}^m_\Sigma))$ is continuous.*

Finally, we give a theorem to show that the ideal interpretation $\mathtt{MR}_{\Pi}$ exists by applying the Tarski's fixed point theorem, which guarantees the existence of the least fixed point and is proven in the reusable part of CSP-Prover.

Theorem 3.3 *The interpretation* $\mathtt{MR}_{\Pi} = \mathtt{LFP}([\![\mathtt{PNfun}_{\Pi}]\!]^{fun}_{\mathcal{R}})$ *exists.*

Proof. By Tarski's fixed point theorem, the function $[\![\mathtt{PNfun}_{\Pi}]\!]^{fun}_{\mathcal{R}}$ has the least fixed point $\bigsqcup\{([\![\mathtt{PNfun}_{\Pi}]\!]^{fun}_{\mathcal{R}})^{(n)}(\bot) \mid n \in \mathit{Nat}\}$ because $[\![\mathtt{PNfun}_{\Pi}]\!]^{fun}_{\mathcal{R}}$ is continuous by Lemma 3.2 and $(\Pi \to \mathit{domR}^{m}_{\Sigma})$ is a cpo with the bottom element $\bot$. □

4 Validation of algebraic laws

Algebraic laws are the core of the proof infrastructure in CSP-Prover. However, they also suit as a means of verification of the semantic clauses: the external choice operator, for example, is supposed to be idempotent, i.e., the equation $P \;\Box\; P =_{\mathcal{R}} P$ shall hold in the stable revivals model. The following Isabelle/HOL code proves this law, over all domains and in both versions semantic of the semantic clauses:

```
lemma cspR_Ext_choice_idem_p:   "P : procR ==> P [+] P =R[M,M] P"
apply (simp add:  cspR_cspT_semantics)
apply (intro conjI)
apply (rule order_antisym)
apply (rule, simp add:  in_deadlocks)
apply (elim conjE exE disjE, simp_all)
apply (rule, simp add:  in_deadlocks)
apply (rule order_antisym)
apply (rule, simp add:  in_revivals)
apply (elim conjE exE disjE, simp_all)
apply (rule, simp add:  in_revivals, force)
done
```

Besides this law (□-idem), we proved in Isabelle/HOL that the following basic laws and step laws hold over all domains and in both versions of the semantic clauses: (⊓-idem), (□-sym), (⊓-sym), ($|[X]|$-sym), (□-assoc), (⊓-assoc), (□-⊓-dist), (Stop-$|[X]|$), ($;$-step), (prefix-step), and ($\downarrow$-step).

Over all domains, however, the standard step law for *Stop*

$$\mathit{Stop} =_{\mathcal{R}} \;?\,x : \emptyset \to P(x)$$

fails, as in Sem^{orig} we have $\langle\rangle \in \mathit{deadlocks}_M(\mathit{Stop})$ but $\mathit{deadlocks}_M(? : \emptyset \to P) = \{\}$. For the same reason also the law (□-step)

$$\begin{array}{l}(?\,x : A \to P(x)) \;\Box\; (?\,x : B \to Q(x)) =_{\mathcal{R}}\\ ?\,x : (A \cup B) \to \mathit{if}\ (x \in A \cap B)\ \mathit{then}\ (P(x) \sqcap Q(x))\\ \qquad\qquad\qquad\quad \mathit{else}\ \mathit{if}\ (x \in A)\ \mathit{then}\ P(x)\ \mathit{else}\ Q(x)\end{array}$$

fails, take the process $?\,x : \{a\} \to \;P(x) \;\Box ?\,x : \{\} \to \;Q(x)$ as a counter example.

Taking our new semantic clauses *Sem*, we have proven in Isabelle/HOL that both these laws as well as all laws listed above are valid over domR_{arb}, and domR^{m}.

Changing the deadlock clause of the prefix choice operator, has consequences for the step law ($[[r]]$-step) for renaming:

$$\begin{array}{l}(?\,x : A \to P(x))[[r]] =_{\mathcal{R}}\\ ?\,x : \{x \mid \exists\, a \in A.\, (a, x) \in r\} \to (!\ a : \{a \in A \mid (a, x) \in r\} \bullet (P(a)[[r]]))\end{array}$$

Taking the new clause for the prefix choice operator with the original clause for the renaming operator fails to hold in the deadlock component, e.g., for the process $(?x : \{a\} \rightarrow \mathit{Skip})[[\emptyset]]$. As demonstrated in Section 3.3., $\mathit{deadlocks}_M^{orig}(\mathit{lhs}) = \emptyset$. The rhs of this step law evaluates to $?x : \emptyset \rightarrow \ldots$, i.e., the modified deadlock clause of the prefix choice operator includes the empty trace $\langle\rangle$. With our new semantic clauses, however, also this law holds over domR, domR_{arb}, and domR^m – also for renaming involving infinite relations.

As expected, we also could prove in Isabelle/HOL (over all domains and with both semantic variants) by giving a counter example that internal choice does not distribute over external choice: $(P \,\Box\, Q) \sqcap R \neq_{\mathcal{R}} (P \sqcap R) \,\Box\, (Q \sqcap R)$.

5 Conclusion

In this paper, we have presented an implementation of different variants of the stable revivals model $\mathcal{R}$ using the proof infrastructure provided by Csp-Prover. Overall, our implementation provides a mechanical verification in Isabelle/HOL that the design of $\mathcal{R}$ as given in [17] has the desired properties, namely that the operators are type correct and continuous. In order to obtain certain algebraic laws, however, we suggest to modify two semantical clauses. We also extend the model $\mathcal{R}$ to infinite alphabets of communications: Here, we suggest to restrict the semantical domain by a stricter healthiness condition which ensures type correctness for infinite renamings.

As future work, we intend to verify further process algebraic laws, especially concerning the parallel operator. We further plan to apply our implementation to practical applications such as the on-line shopping example given in [19].

Acknowledgement

We would like to thank Bill Roscoe for his valuable feedback on our implementation, as well as Erwin R. Catesbeiana (jr) for advice on the verification of component based systems.

References

[1] A.E. Abdallah, C.B. Jones, and J.W. Sanders, editors. *CSP: The First 25 Years*, LNCS 3525. Springer, 2005.

[2] A. J. Camilleri. Mechanizing CSP trace theory in higher order logic. *IEEE Transactions on Software Engineering*, 16(9):993–1004, 1990.

[3] A. J. Camilleri. A higher order logic mechanization of the CSP failure-divergence semantics. In *Higher Order Workshop*. Springer, 1991.

[4] B. Dutertre and S. Schneider. Using a PVS embedding of CSP to verify authentication protocols. In *TPHOL'97*. Springer, 1997.

[5] Failures-divergence refinement. User Manual, Obtainable from http://www.fsel.com/fdr2_manual.html.

[6] C. A. R. Hoare. *Communicating Sequential Processes.* Prentice Hall, 1985.

[7] Y. Isobe and M. Roggenbach. A generic theorem prover of CSP refinement. In *TACAS 2005*, LNCS 3440, pages 108–123. Springer, 2005.

[8] Y. Isobe and M. Roggenbach. A complete axiomatic semantics for the CSP stable-failures model. In *CONCUR 06*, LNCS 4137, pages 158–172, 2006.

[9] Y. Isobe and M. Roggenbach. CSP-Prover – a proof tool for the verification of scalable concurrent systems. *JSSST (Japan Society for Software Science and Technology) Computer Software*, 25, 2008.

[10] Y. Isobe and M. Roggenbach. Proof principles of CSP – CSP-Prover in practice. In *LDIC 2007*. Springer, 2008.

[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] F. Kammueller. CSP revisited. Technical Report 364/07, Department of Computer Science, University of Kaiserslautern, 2007.

[13] T. Nipkow, L. C. Paulson, and M. Wenzel. *Isabelle/HOL — A Proof Assistant for Higher-Order Logic*. LNCS 2283. Springer, 2002.

[14] S. Owre, J. M. Rushby, and N. Shankar. PVS: A prototype verification system. In *CADE-11*. Springer, 1992.

[15] A. W. Roscoe. *The Theory and Practice of Concurrency*. Prentice Hall, 1998.

[16] A.W. Roscoe. Private conversation with M. Roggenbach, 2006.

[17] A. W. Roscoe. Revivals, stuckness and the hierarchy of CSP Models. Revision of the 2005 draft, December 2007.

[18] A. W. Roscoe, J. N. Reed, and J. E. Sinclair. Responsiveness of interoperating components. *Formal Aspects of Computing*, 16:394–411, 2004.

[19] A. W. Roscoe, J. N Reed, and J. E Sinclair. Machine-verifiable responsiveness. In *Proceedings of AVOCS 2005*, 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] D. G. Samuel. Implementing the stable revivals model. Master's thesis, Swansea University, 2008.

[22] D. G. Samuel, Y. Isobe, and M. Roggenbach. Reasoning on Responsiveness – Extending CSP-Prover by the Model R. In *NWPT'07/FLACOS'07*, Reseach Report 366. Oslo University, 2007.

[23] H. Tej and B. Wolff. A corrected failure divergence model for CSP in Isabelle/HOL. In *FME '97*. Springer, 1997.

[24] K. Wei and J. Heather. Embedding the Stable Failures Model of Csp in PVS. In *International Conference on Integrated Formal Methods*, LNCS 3771. Springer, 2005.