Extending CSP-Prover by deadlock-analysis: Towards the verification of systolic arrays

Yoshinao Isobe* **Markus Roggenbach†** **Stefan Gruner‡**

Summary. CSP-Prover provides a deep encoding of the process algebra CSP in the interactive theorem prover Isabelle. Here, we extend CSP-Prover by a framework for the deadlock-analysis of networks. As a typical example we study systolic arrays and prove in CSP-Prover that Kung's classical algorithm for matrix-multiplication is deadlock-free.

1 Introduction

Among the various frameworks for the description and modeling of reactive systems, *process algebra* plays a prominent rôle. It has turned out to be suitable for requirement specifications, design specifications, and also for formal refinement proofs [3]. In this context, the process algebra CSP [8, 18] has been successfully applied in various areas such as train control systems [6], software for the International Space Station [4, 5], or verification of security protocols [20].

CSP-Prover [9, 10] provides a deep encoding of the process algebra CSP within the interactive theorem prover Isabelle [16]. Its approach of interactive theorem proving complements proofs by (finite) model checking in FDR [13]. The strength of CSP-Prover is to be found in the analysis of large or even infinite state systems as well as in its ability to deal with parametrised systems such as systolic arrays.

Deadlocks are certainly the best known and also most feared failures exhibited by concurrent systems. For parallel algorithms such as systolic arrays, the proof of deadlock-freedom is as fundamental as the proof of termination for sequential algorithms. Systolic arrays, see e.g. [17], often deal with loosely specified data such as matrices over rings. Furthermore, they scale up with the size of the problem, e.g. Kung's systolic array for the multiplication of $n \times n$ matrices requires n^2 processing elements. For these reasons it is impossible to treat such systems solely within the classical model checking approach that requires systems to be finite state.

Proofs of deadlock-freedom in CSP-Prover alone [10] (e.g. that the interaction of certain components of the electronic payment system ep2, a new international standard for electronic payment systems, is deadlock-free) were based on the facts that (i) there are CSP processes that are deadlock-free by construction and (ii) that CSP's stable-failures refinement is deadlock-preserving.

In this paper we show how to extend CSP-Prover with Deadlock-Freedom Proof Package (abbreviated to DFP) which provides a proof technique[1] suggested by Roscoe & Dathi [19] in order to analyse networks. For example, DFP contains a theorem for localizing the proof of deadlock freedom of a whole systolic array to the analysis of each element with its neighbors. We apply CSP-Prover with DFP to verify that Kung's algorithm for matrix multiplication [12] is deadlock-free.

*National Institute of Advanced Industrial Science and Technology, Japan.

†University of Wales Swansea, United Kingdom.

‡Swansea Institute to the University of Wales, United Kingdom.

[1] It was slightly modified from [19] to fulfill the needs of a semi-automatic proof technique.

2 CSP and CSP-Prover

Syntax and semantics of the process algebra Csp [8,18] are defined relative to a given set of communications. Its *basic processes* are built from *primitive processes* such as `SKIP` (successful termination) or `STOP` (deadlock). Csp includes communication primitives for sending and receiving values over a (perfect, non-lossy) communication channel, distinguishes between internal and external choice between two processes, and offers a variety of parallel operators, sequential composition of processes, as well as various other features such as renaming and hiding. *Recursive processes* are either defined by process equations or by so-called μ-recursion.

Csp is a language with many semantics, different in their style as well as in their ability to distinguish between various forms of reactive behaviour. There are operational, denotational and algebraic approaches, ranging from the simple finite traces model $\mathcal{T}$ to such complex semantics as the infinite traces model with failures/divergences $\mathcal{U}$. For theorem proving the denotational semantics are especially interesting. Here, we concentrate on the stable-failures model $\mathcal{F}$, which is the Csp semantics best suited for deadlock-analysis.

Given a set of communications A, the domain of the *stable failures model* $\mathcal{F}$ is a set of pairs (T, F) satisfying certain healthiness conditions (p. 208 in [18]), where $T \subseteq A^{*\checkmark}$ and $F \subseteq A^{*\checkmark} \times \mathbb{P}(A^{\checkmark})$[2]. In such a pair (T, F), T is the set of traces a process can execute, while the elements $(s, X) \in F$ describe sets of communications X which a process can fail to engage in after executing the trace s. For example, the semantic clauses of $\mathcal{F}$ for the Csp action prefix $a \rightarrow P$ is given as follows:

$$\begin{array}{c} \mathit{traces}(a \rightarrow P) = \{\langle\rangle\} \cup \{\langle a\rangle \frown s \mid s \in \mathit{traces}(P)\} \\ \mathit{failures}(a \rightarrow P) = \{(\langle\rangle, X) \mid a \notin X\} \cup \{(\langle a\rangle \frown s, X) \mid (s, X) \in \mathit{failures}(P)\} \end{array}$$

which means $a \rightarrow P$ can perform only a and thereafter behaves like P.

The domain $\mathcal{F}$ can be turned into both a complete partial order and a complete metric space. This allows to analyse Csp's recursive process equations by the fixpoint theorems of Tarski and Banach, respectively. Both these fixpoint theorems give rise to powerful fixpoint induction techniques.

Csp-Prover [9,10] provides a faithful implementation of the *stable failures model* $\mathcal{F}$ of [18] based on the higher order logic HOL [15] with complex numbers of the interactive theorem prover Isabelle [16]. To extend an existing logic such as HOL-Complex, Isabelle offers mechanisms to define new types, functions, predicates etc. It is possible to establish new theorems which are entered as *goals*. A goal can be manipulated by proof commands referring to a set of predefined inference *rules* producing new goals. Such rules can be combined to form *proof tactics*. A proof is completed if by application of rules and tactics the only open goal is the truth value `True`. Successfully proven theorems can be stored and used later as new rules. Theorems, together with definitions and proof commands needed for their proofs, can be stored in *theory files*. Isabelle organises such files in a database, to which other theory files may refer.

Fig. 1 shows the generic architecture of Csp-Prover instantiated with the stable failures model $\mathcal{F}$. On top of the logic HOL-Complex there is a large, model-independent, *reusable part*. This contains Banach's fixpoint theorem and the metric fixpoint induction rule based on complete metric spaces as well as Tarski's

[2] $A^{\checkmark} := A \cup \{\checkmark\}$, $A^{*\checkmark} := A^* \cup \{s \frown \langle\checkmark\rangle \mid s \in A^*\}$. $\checkmark \notin A$ is a special event denoting successful termination.

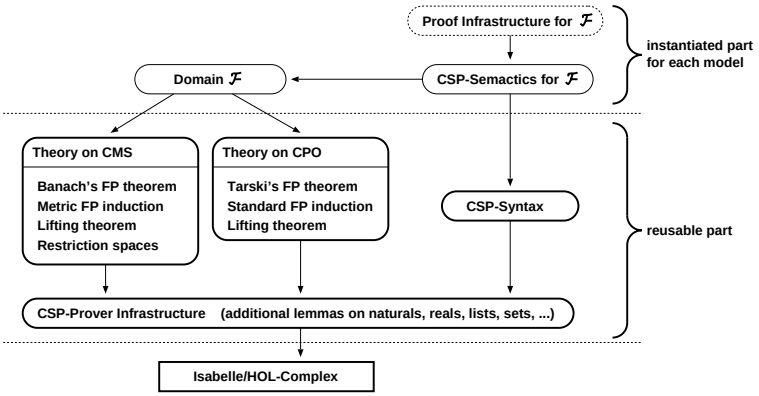

Fig.1 The theory map of CSP-Prover instantiated with the stable-failures model $\mathcal{F}$.

fixpoint theorem and the standard fixpoint induction rule based on complete partial orders. Furthermore, infinite product constructions are provided as lifting theorems which are required for infinite state systems. Another contribution of the model-independent part is the definition of the Csp syntax as a recursive type. This means that the syntax is *deeply encoded*. Therefore, Csp-Prover supports structural induction on processes. The model-dependent, *instantiated part* of Csp-Prover is shown here for the stable failures model $\mathcal{F}$. It defines the domain $\mathcal{F}$ together with proofs in Isabelle which establish that $\mathcal{F}$ indeed is a complete metric space as well as a complete partial order. Furthermore, it defines the semantic clauses of $\mathcal{F}$. Finally, model-dependent proof infrastructure is provided, such as step laws as well as distributive laws holding in $\mathcal{F}$, or the tactic `csp_hnf_tac` that translates Csp expressions into Head Normal Form. As these laws and tactics are proven in Isabelle they are guaranteed to be sound with respect to $\mathcal{F}$.

3 Proving deadlock-freedom of CSP networks

A deadlock is a state which refuses to engage in any event. Thus, given a set of communications A, a process P is said to be *deadlock-free* iff: $\forall\, s \in A^*.\ (s, A^{\checkmark}) \notin \mathit{failures}(P)$, i.e. after every possible trace s there will always be at least one event the process P cannot refuse to engage in.

A *network* is a special way of defining a process in Csp [19]. Formally, a network V is a finite set of pairs $\{(P_i, X_i) \mid i \in I\}$, where I is a nonempty, finite index set, P_i is a Csp process, and $X_i \subseteq A$ is the set of communications which P_i can engage in, for all $i \in I$. The process defined by such a network V is

$$PAR(V) := \|_{i\in I}\,(P_i, X_i)$$

where $\|_{i\in I}\,(P_i, X_i)$ is the replicated alphabetized parallel operator of Csp. Each process P_i can perform $a \in X_i$. P_i and P_j can communicate with $a \in X_i \cap X_j$. As the semantics of $\|_{i\in I}$ is independent of the order of its arguments, it is sufficient to define networks as sets.

A network is called *triple disjoint* iff $X_i \cap X_j \cap X_k = \emptyset$ whenever i, j and k are distinct, i.e. all communications in the network are point to point. We call a network *busy* iff all of its component processes P_i are deadlock free. The *state of a network* is a pair $\sigma = (s, (Y_i)_{i\in I})$ such that $(s \restriction X_i, Y_i) \in \mathit{failures}(P_i)\backslash X_i$ [3] for all $i \in I$, i.e. the process P_i refuses Y_i after engaging in the trace s reduced to its

[3] $F\backslash X := \{(s, Y) \in F \mid Y \subseteq X\}$. Note that P_i cannot engage in $a \notin X_i$ in $\|_{i\in I}\,(P_i, X_i)$.

individual set of communications X_i. Let $\Lambda \subseteq A$ be a set of communications. An *ungranted request in a state σ with respect to* Λ, written $V \triangleright i \xrightarrow{\sigma,\Lambda}\bullet\ j$, is a pair of indices (i,j) with $i \neq j$ such that process P_i wants to communicate with process P_j (i.e. $(X_i - Y_i) \cap X_j \neq \emptyset$) but all communications common to P_i and P_j can be refused (i.e. $X_i \cap X_j \subseteq Y_i \cup Y_j$), and $(X_i - Y_i) \cup (X_j - Y_j) \subseteq \Lambda$. For triple-disjoint networks, the set Λ is often chosen to be the *vocabulary*, given by $\bigcup\{X_i \cap X_j \mid i \neq j\}$, of the network. Using these notions, Roscoe and Dathi [19] prove:

Theorem 3.1 *Let $V = \{(P_i, X_i) \mid i \in I\}$ be a triple disjoint, busy network with vocabulary Λ. Let $(D, \leq)$ be a partial order. Let $f_i : \overline{failures(P_i)\backslash X_i} \to D,\ i \in I$ be functions* [4] *with the following property: for all states $\sigma = (s, (Y_i, Y_j))$ of all two-element networks $V' = \{(P_i, X_i), (P_j, X_j)\}$ $(i \neq j)$ that can be formed by selecting two elements of V holds: if $V' \triangleright i \xrightarrow{\sigma,\Lambda}\bullet\ j$, then $f_i(s \restriction X_i, Y_i) > f_j(s \restriction X_j, Y_j)$. Then $PAR(V)$ is deadlock free.* □

This theorem localizes the proof of deadlock freedom for the whole network consisting of $|\,I\,|$ processes to the analysis of the $(|\,I\,| * (|\,I\,| -1))/2$ networks which consist of two different processes only. Roscoe and Dathi [19] successfully apply this kind of reasoning to a wide class of networks, including systolic arrays, a resource allocation protocol, and the dining philosophers.

4 Systolic arrays

Systolic arrays are parallel algorithms that *"typically consist of a large number of similar processing elements which are interconnected to exchange data"* where it is characteristic that (1) interconnections are local only, (2) data moves at a constant velocity, and (3) each of the elements performs just a certain part of the computation required to solve the problem [17]. Well performing systolic arrays have been formulated in areas so different as linear algebra (e.g. matrix multiplication, solving systems of linear equations, computing the inverse of a matrix), sorting, dynamic programming, and computational geometry. Applications include numerics, signal processing, and computer graphics, where systolic arrays are often realised in VLSI or as Field Programmable Gate Arrays, see e.g. [11,22].

Usually, systolic arrays are formulated with the help of recurrences. Consider for example the multiplication of two square matrices A and B over a ring R of dimension $n \in \mathbf{N}$ defined as usual by

$$(C)_{i,j} = (AB)_{i,j} := \textstyle\sum_{k \in I_n} A_{i,k} B_{k,j}\,, \quad \forall\ i,j \in I_n,\ I_n := \{0, \ldots, n-1\}.$$

With recurrences defined as

Input equations	$a_{i,0,k} := A_{i,k}$	$\forall\, i,k \in I_n$
	$b_{0,j,k} := B_{k,j}$	$\forall\, j,k \in I_n$
Moving A, B	$a_{i,j+1,k} := a_{i,j,k} \quad \forall\, i,k \in I_n,$	$\forall\, j \in I_n \backslash \{n-1\}$
	$b_{i+1,j,k} := b_{i,j,k} \quad \forall\, j,k \in I_n,$	$\forall\, i \in I_n \backslash \{n-1\}$
Initialising C	$c_{i,j,0} := a_{i,j,0} b_{i,j,0}$	$\forall\, i,j \in I_n$
Computing C	$c_{i,j,k+1} := c_{i,j,k} + a_{i,j,k+1} b_{i,j,k+1}$	$\forall\, i,j \in I_n, \forall\, k \in I_n \backslash \{n-1\}$

we obtain $\forall\, i,j \in I_n.\, c_{i,j,n-1} = (C)_{i,j}$. Allocating (1) the n computations described by the recurrences $c_{i,j,k}$ for $k = 0, ..., n-1$ to *one* processing element $pe_{i,j}$ for $i,j \in I_n$, and (2) interpreting the moving equations as communications, we obtain the systolic algorithm introduced by Kung [12] for matrix multiplication on an orthogonal and quadratic array of processing elements. Fig. 2 shows the allocation of the computing recurrences to processing elements for $n = 3$. For

[4] $\overline{F}$ is defined as: $(s, X) \in \overline{F} :\Leftrightarrow X = \max\{Y \mid (s, Y) \in F\}$.

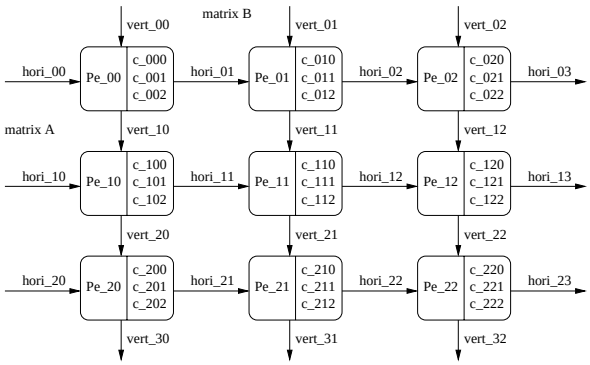

Fig.2 Allocations of recurrences to processing elements

each processing element pe_{ij} it collects the indices i, j, k for those computations $c_{i,j,k}$ that this specific *pe* shall perform. The arrows indicate how the matrices A and B flow through the network. The matrix A moves over the connections *hori* from left to the right, the matrix B flows over the connections *vert* from top to bottom. Other systolic algorithms for matrix multiplication work, e.g., with an orthogonal rectangular or with a hexagonal interconnection network.

We formalize the behaviour of the single processing elements $(pe(i,j)\,r)$ of Kung's matrix multiplication algorithm in Csp. Their behaviour is defined by recursive process equations for a function f:

$$\begin{array}{rl} f(pe(i,j)\,r) = & (vert(i,j)?a \to hori(i,j)?b \to (f(pe'(i,j)\,r\,a\,b))) \;\Box \\ & (hori(i,j)?b \to vert(i,j)?a \to (f(pe'(i,j)\,r\,a\,b))) \\ f(pe'(i,j)\,r\,a\,b) = & ((vert(i+1,j)!a \to \texttt{SKIP}) \;|||\; (hori(i,j+1)!b \to \texttt{SKIP}))\;{}_{;} \\ & (f(pe(i,j)(r + a * b))) \end{array}$$

Here, *vert* and *hori* are arrays of Csp channels. Any channel, take for instance $hori(i,j)$, can transport arbitrary ring elements r. The processing element *pe* in position (i,j) initially holds an intermediate value $r \in R$ for the matrix C. It then receives values a and b from its upper vertical and left horizontal connection in either order. Then it behaves like the process $pe'(i,j)$ which first sends the values a and b over its lower vertical and right horizontal connection and then updates the value held in $pe(i,j)$ to $(r + a * b)$. As f is a (higher order) process itself, the existence and uniqueness of a solution of the above recursive process definition follows by Banach's fixpoint theorem.

While the derivation of Kung's algorithm might be convincing enough to claim that it always will produce a correct result, it is by no means obvious if its precise formulation in the framework provided by Csp is deadlock free for all dimensions n. Define sets of communications $Com(i,j)$, for every $i, j \in I_n$, as

$$Com(i,j) := \{vert(i,j)\,r, vert(i+1,j)\,r, hori(i,j)\,r, hori(i,j+1)\,r \;\mid\; r \in R\}.$$

With these notions we can define the Csp network describing Kung's systolic array for matrix multiplication as $V_{mul} := \{(f(pe(i,j),0), Com(i,j)) \mid i \in I_n \wedge j \in I_n\}$.

In the rest of the paper we will show how to verify that the network V_{mul} is deadlock-free for all dimensions n. Note the high degree of parametrisation of this claim: The first parameter is the ring R over which the matrix multiplication is defined. The second parameter is the dimension of the matrices that determines the number of processing elements involved. It is this high degree of parametrisation which makes interactive theorem proving the appropriate technique.

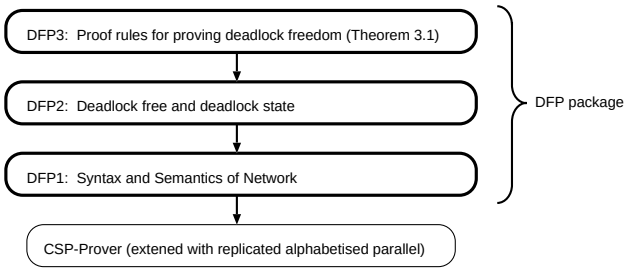

Fig.3 The structure of the package DFP.

5 Implementation of a DFP package on top of CSP-Prover

Csp-Prover provides an open framework that allows one to analyse and verify Csp-processes. In order to extend Csp-Prover by syntactic constructions and verification techniques specific to a certain problem class, new Isabelle theories have to be defined. In the case of deadlock analysis of networks such as our example V_{mul} above we first need to define syntax and semantics of networks (package DFP_1), then to give characterisations for deadlock (package DFP_2), and finally to provide semi-automatic techniques for deadlock analysis (package DFP_3), see Fig. 3 for the structure of the whole Deadlock-Freedom Proof Package.

The mechanisms to define Csp networks as described in Sect. 3 are provided in the Isabelle theory DFP_1:

```
types ('i,'a) Network = "('i set * ('i => ('a proc * 'a set)))"
consts PAR :: "('i,'a) Network => 'a proc"
defs   PAR_def : "PAR V == [||] i:(fst V) ((snd V) i)"
syntax "@Network" :: "('a proc * 'a set) => pttrn => 'i set
                      => ('i,'a) Network" ("{ _ | _:_ }net")
translations "{ PX | i:I }net" == "(I, (λi. PX))"
```

The Isabelle keyword **types** defines type synonyms. An `('i,'a) Network` is a pair consisting of an index set and map from indices to pairs, which consist of a process and a set of communications. Here, `'i` and `'a` are type parameters for the index set and the set of communications, resp. `set` is the Isabelle type for sets, while `proc` is Csp-Prover's type for Csp-processes. **consts** declares function types which are then defined by **defs**. We use this mechanism here in order to define the process of a network, c.f. Sect. 3. Given a network `V`, its process `(PAR V)` is obtained by applying the replicated alphabetised parallel operator `[||]` to its two components. These are obtained by the projection functions `fst` and `snd`, resp. **syntax** and **translations** can be used to introduce syntactic sugar. With the above definitions, for example, the network `(I, (λi. (i -> SKIP, {i}))` can be formulated simply as `{ (i -> SKIP, {i}) | i:I }net`.

DFP_2 collects characterisations of deadlock as well as general infrastructure. A typical example is the following lemma: A network `{(P i,X i)| i:I}net` is in a deadlock after some trace `t` if $\bigcup_{\texttt{i}\in\texttt{I}}$`(Y i)` = $\bigcup_{\texttt{i}\in\texttt{I}}$`(X i)` for some `Y` such that for all `i`$\in$`I`, `(t`$\restriction$`(X i), Y i)`$\in$ *failures*`(P i)`$\setminus$`(X i)`. To formulate this lemma, it is convenient to complement the *syntactical* network `{(P i,X i) | i:I}net` by a *semantical* network `{(F i,X i) | i:I}Fnet`. Here, `(F i,X i)` are pairs consisting of an indexed failure set and an indexed set of communications. Given a syntactic net `{(P i,X i) | i:I}net`, the simplest semantical network that we can associate with it is {(*failures*`(P i)`$\setminus$`(X i), X i) | i:I}Fnet`. This makes

sense as a process `(P i)` never can engage in events that are not contained in its communications `X i`. But *failures*`(P i)` contains many failures useless from the perspective of this kind of deadlock analysis: if `(s,Y)`$\in$ *failures*`(P i)` and `Y'`$\subseteq$`Y`, then also `(s,Y')`$\in$ *failures*`(P i)`. This makes it difficult to find and define the functions f_i of the above stated Theorem 3.1. To this end, Roscoe and Dathi work with the maximum $\overline{\textit{failures}(\texttt{P i})\backslash(\texttt{X i})}$. However, it is hard to automatically prove that a certain set of failures is indeed the maximum. This difficulty arises in the context of non-determinism. Also, the potential infinity of the set *failures*`(P i)` plays here a rôle. Therefore, we re-formulate the theory developed by Roscoe and Dathi at this point. We take those failures `(F i)` into the semantical network associated with a syntactical network `{(P i,X i)| i:I}net`, for which the relation `(F i)` $\subseteq_\exists$ *failures*`(P i)`$\backslash$`(X i)` holds for all `i`$\in$`I`. Here, we define `F` $\subseteq_\exists$ `E` by

`(*) F` $\subseteq$ `E` $\wedge$ $\forall$`(s,Y)`$\in$`E.` $\exists$`Z. (s,Z)`$\in$`F` $\wedge$ `Y`$\subseteq$`Z`

Note that *failures*`(P)`$\backslash$`X` $\subseteq_\exists$ *failures*`(P)`$\backslash$`X` and $\overline{\textit{failures}(\texttt{P})\backslash\texttt{X}}$ $\subseteq_\exists$ *failures*`(P)`$\backslash$`X`, which are extreme cases to satisfy the condition `(*)`.

Based on these notions, in DFP$_3$ we prove a modified Theorem 3.1 in CSP-Prover, where the weaker condition `(*)` rather than the maximum one is applied:

```
theorem Rule1_Roscoe_Dathi_1987:
"[|VF = {(F i, X i) | i:I}Fnet; V = {(P i, X i) | i:I}net;
   VF isFailureOf V; I ≠ {}; finite I;
   triple_disjoint VF; BusyNetwork VF ;
   ∃f::('i => 'a failure => ('pi::order)). ∀i:I. ∀j:I. i ≠ j -->
   {(F k, X k) | k:{i,j}}Fnet >> i --[(t,Y),(VocabularyOf VF)]-->o j
   --> f j (t rest-tr (X j), Y j) < f i (t rest-tr (X i), Y i)) |]
 ==> DeadlockFree (PAR V)"
```

If one applies this theorem to a network, the main proof obligations are (1) to find a function `f` and (2) to prove the inequality `<`. Here, some experience is required to find such function, but CSP-Prover can assist the proofs.[5]

To discharge the proof obligation `VF isFailureOf V`, i.e. to show for each `i`$\in$`I` that `(F i)` $\subseteq_\exists$ *failures*`(P i)`$\backslash$`(X i)`, where `(X i)` is the set of communications in which the process `(P i)` can engage in, the package DFP$_1$ provides a set of assisting lemmas. For instance, if a process `(P i)` is of the the form `(? x:X -> Pf x)` (external prefix choice), the following result can be applied:

```
lemma subex_Ext_pre_choice:
"[| ∀ a:X. Ff a ⊆∃ failures(Pf a) ;
  F = {([]t, A-Ev'X)}∪{([Ev a]_t @_t s, Y)|a s Y. (s,Y):Ff a ∧ a:X} |]
==> F ⊆∃ failures(? x:X -> Pf x)\A"
```

i.e. there is a certain set of failures `F` below the failures of `(P i)` with respect to $\subseteq_\exists$, which is constructed based on the failures of the simpler processes `(Pf x)`.

The package DFP$_1$ contains also lemmas for (replicated) internal choice. There is no need to support further syntactic patterns as CSP-Prover's tactic `csp_hnf_tac` fully automatically translates CSP expressions into Head Normal Form (external prefix choice or (replicated) internal choice). Thus, `(F i)` $\subseteq_\exists$ *failures*`(P i)`$\backslash$`(X i)` can be proven by structural induction over `(P i)`.

6 Application

In this section we demonstrate how CSP-Prover extended by the DFP package can be used to prove deadlock freedom of Kung's matrix multiplication algorithm

[5] We will extend the package with additional rules to find such a function.

by applying the theorem `Rule1_Roscoe_Dathi_1987`. Although the definitions as well as the proof details are specific to the Kung's algorithm, their style carries over to the other systolic arrays and provides a blueprint for further proofs.

Step 1: The processing elements `pe(i,j)` and the network are defined:

```
recdef Def "{}"
"Def(X,(pe(i,j) r))      =(vert(i,j)?x -> hori(i,j)?y -> (X(pe'(i,j) r x y)))[+]
                          (hori(i,j)?y -> vert(i,j)?x -> (X(pe'(i,j) r x y)))"
"Def(X,(pe'(i,j) r x y))=((vert(i+1,j)!x -> SKIP)|||(hori(i,j+1)!y -> SKIP));;
                          (X(pe(i,j)(r + x * y)))"
recdef Com "{}"
  "Com(i,j) = {a. ∃x. (a = vert(i,j) x) ∨ (a = vert(i+1,j) x)
                   ∨ (a = hori(i,j) x) ∨ (a = hori(i,j+1) x)}"
defs Fun_def : "Fun    == curry Def"
     Fix_def : "Fix p  == (FIX Fun) p"
     Vmul_def: "Vmul N == {(Fix(pe(i,j) 0),Com(i,j)) | (i,j):{(i,j). i<N∧j<N}}net"
```

`curry` is a function for transforming non-curried functions to curried ones. `(FIX Fun)` denotes the fixpoint of `Fun`. The Isabelle's keyword **recdef** allows to define recursive processes in a similar way to the conventional definition of CSP[6] (c.f. *pe* in Sect. 4). In this case, the recursive process for `(pe(i,j) r)` is given by `Fix(pe(i,j) r)`, and the network is defined as shown at the bottom line.

Step 2: A semantical network `VmulF` of the syntactical network `Vmul` is defined.

```
defs
peFin_def: "peFin == (λ(i,j) F.
 {([]t, Com(i,j)-({hori(i,j+1) y |y. True}∪{vert(i+1,j) x |x. True}))}∪
 {([Ev(vert(i+1,j) x)]t, Com(i,j)-{hori(i,j+1) y |y. True}) |x. True} ∪
 {([Ev(hori(i,j+1) y)]t, Com(i,j)-{vert(i+1,j) x |x. True}) |y. True} ∪
 {([Ev(vert(i+1,j) x),Ev(hori(i,j+1) y)]t @t s, Y) |x y s Y. (s,Y):eqFout(i,j) x y F}∪
 {([Ev(hori(i,j+1) y),Ev(vert(i+1,j) x)]t @t s, Y) |x y s Y. (s,Y):eqFout(i,j) x y F})"
peFout_def: "peFout == (λ(i,j) x y F.
 {([]t, Com(i,j)-{hori(i,j+1) y, vert(i+1,j) x})} ∪
 {([Ev(vert(i+1,j) x)]t, Com(i,j)-{hori(i,j+1) y})} ∪
 {([Ev(hori(i,j+1) y)]t, Com(i,j)-{vert(i+1,j) x})} ∪
 {([Ev(vert(i+1,j) x),Ev(hori(i,j+1) y)]t @t s, Y) |s Y. (s,Y):F} ∪
 {([Ev(hori(i,j+1) y),Ev(vert(i+1,j) x)]t @t s, Y) |s Y. (s,Y):F})"
primrec "peF_rec 0 = (λ(i,j). {})"
        "peF_rec (Suc n) = (λ(i,j). peFin (i,j) (peF_rec n (i,j)))"
defs peF_def : "peF ij == ⋃ {peF_rec n ij |n. True}"
     VmulF_def: "VmulF N == {(peF(i,j), Com(i,j)) | (i,j):{(i,j). i<N∧j<N}}Fnet"
```

The failures set `peF(i,j)` is defined inductively on the number of iterations by Isabelle's keyword **primrec**. Intuitively, it means that at first all the events in `Com(i,j)` are refused except `hori(i,j+1)` and `vert(i+1,j)`, then events except `hori(i,j+1)` are refused after `vert(i+1,j)` or events except `vert(i+1,j)` are refused after `hori(i,j+1)`. This represents the meanings of `pe(i,j)` well.

Step 3: It is proven that `VmulF` is a semantic network of `Vmul`:

```
lemma ex_matrix_isFailureOf: "(VmulF N) isFailureOf (Vmul N)"
```

By unfolding definitions of `isFailureOf`, `VmulF`, and `Vmul`, the goal can be rewritten to: `"∀i<N. ∀j<N. peF(i,j)` $\subseteq_{\exists}$ *failures*`((FIX Fun) pe(i,j) 0)"`. Here, the fixpoint `(FIX Fun)` can be equal to the replicated internal choice among iterated compositions $\texttt{Fun}^{(\texttt{n})}(\bot)$ over all `n` by Tarski's fixpoint theorem, where $\bot$ is a higher order process (`λx. DIV`) whose meaning is the bottom element in the model $\mathcal{F}$. Therefore, the subgoal can be shown by proving the following lemma:

[6] In **recdef**, pattern matching is done only for the first argument. Therefore, a pair argument in `Def` is used, then it is transformed by `curry` in `Fun`.

lemma sub: "peF_rec n (i,j) $\subseteq_\exists$ *failures*(Fun$^{(2*n)}$(⊥) pe(i,j) r)"

Here, 2 in Fun$^{(2*n)}$ means that Fun is unfolded twice a loop (i.e receiving and sending), while peF_rec is unfolded once. This lemma can be proven by induction on n and lemmas (e.g. subex_Ext_pre_choice) given in Section 5.

In fact, Step 2 and Step 3 are performed in parallel because CSP-Prover with DFP are very helpful not only for proving this lemma but also for defining VmulF.

Step 4: For some function f the inequality < defined in Theorem 3.1 need to be shown. In our example, the function (λ(i,j) (s,X). (length$_t$ s)+2*(i+j)) was selected. The proof of the inequality was well assisted by CSP-Prover. Currently, it is difficult to automatically find such a function, but there are certain patterns to find it. It is future work to encode such patterns as rules in DFP and to assist finding such a function.

Step 5: Finally, it is proven that (Vmul N) is deadlock free for any size N:

theorem ex_matrix_DF: "∀ N. N ≠ 0 --> DeadlockFree (PAR (Vmul N))"

Applying the theorem Rule1_Roscoe_Dathi_1987 (i.e. encoded Theorem 3.1) as a proof rule to the goal, the assumptions of the proof rule are displayed as subgoals (i.e. next proof obligations). The important subgoals have already been proven in Steps 3 and 4. The others (i.e. the index set is finite, and VmulF is triple disjoint and a busy network) are much easier than proofs in Steps 3 and 4.

We actually proved the theorem ex_matrix_DF on CSP-Prover with DFP, without any abstraction, which means containing value passing and computation. It can be an infinite system because the type of value is allowed to be any instance (e.g. nat, int, and real) of the type-class ring. This result shows the possibility that CSP-Prover can prove the correctness of computation.

We spent 3 days in describing the proof scripts (i.e. theory files) for proving the theorem ex_matrix_DF, in following the instruction (Steps 1∼5) given in this section. CSP-Prover well assisted us to describe the script by displaying subgoals (e.g. lemmas) for proving main goals (e.g. theorems). The total number of lines in the proof script is 1625. It took about 3 minutes to prove the theorem Rule1_Roscoe_Dathi_1987 by loading the proof script in CSP-Prover with DFP on a personal computer with 1.5GHz CPU (Pentium M). The whole proof script can be downloaded from the web-page [9] of CSP-Prover.

7 Other approaches to verify systolic arrays

The verification of systolic arrays in general is an area less studied than the construction (and, possibly, ad-hoc verification) of particular systolic algorithms.

Margaria et al. [14] provide a verification method for a wider class of systolic systems. Their approach is based on a monadic second order *logic* over the domain of strings. The granularity of their modelling is so fine that it reaches the level of clock-cycles and signals. The authors report "full automation" via their proof tool MOSEL for their case studies on linear (1-dimensional) systolic arrays.

Recent work by Steggles [21] suggests an higher-order approach too; here it is higher-order *algebra* instead of higher-order logic for modelling and verifying families of 2-dimensional systolic systems. Proofs are conducted manually, proof automation in the term-rewriting tool ELAN is considered.

[2] develops a new mathematical framework of so-called linear guarded ring expressions in order to describe and verify a class of systolic arrays (convolution). Automation-related decidability results are stated by the same author in [1]: Systolic matrix multiplication is on the decidable side, whereas other systolic verifi-

cation problems are reducible to Hilbert's 10^{th} Problem and thus undecidable.

The Isabelle-HOL method for verifying hardware-related properties does not only apply to systolic systems: very recent work [7] describes the verification of the I/O behaviour of a processor via Isabelle-HOL.

8 Conclusion and future work

We have extended Csp-Prover [9, 10] by a framework for automated deadlock-analysis of networks. The new package Dfp automatises proofs of deadlock-freedom to a great extend: It provides syntactic mechanisms to define Csp networks, a general proof rule to establish deadlock-freedom, as well as proof tactics to automatically discharge typical proof obligations. As an application we studied the classical example of Kung's systolic array for matrix multiplication and proved its deadlock-freedom with Dfp. Our work aims to the verification of scalable networks, of which systolic arrays are a typical example. Future work will include proofs of the deadlock-freedom of further systolic arrays as well as of other classes of networks. Furthermore, for systolic arrays we want also to address the question of formal correctness proofs in Csp-Prover.

Acknowledgement The authors would like to thank Erwin R. Catesbeiana jr for help with the intricacies of deadlocks.

References

[1] P. A. Abdulla. Decidable and undecidable problems in systolic circuit verification, 1991.
[2] P. A. Abdulla. Automatic verification of a class systolic circuits. *Formal Asp. Comput.*, 4(2):149–194, 1992.
[3] J. A. Bergstra, A. Ponse, and S. A. Smolka. *Handbook of Process Algebra*. Elsevier, 2001.
[4] B. Buth, M. Kouvaras, J. Peleska, and H. Shi. Deadlock analysis for a fault-tolerant system. In *AMAST'97*, LNCS 1349, pages 60–75. Springer, 1997.
[5] 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.
[6] 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.
[7] A. Fox. An algebraic framework for verifying the correctness of hardware with input and output: A formalization in HOL. In *CALCO'05*, LNCS 3629, pages 157–174. Springer, 2005.
[8] C. A. R. Hoare. *Communicating Sequential Processes*. Prentice Hall, 1985.
[9] Y. Isobe and M. Roggenbach. Webpage on Csp-Prover. `http://staff.aist.go.jp/y-isobe/CSP-Prover/CSP-Prover.html`.
[10] Y. Isobe and M. Roggenbach. A generic theorem prover of CSP refinement. In *TACAS 2005*, LNCS 3440, pages 108–123. Springer, 2005.
[11] T. Jebelean. Implementing GCD systolic arrays on FPGA. *LNCS 849*, pages 132–, 1994.
[12] H. T. Kung. The structure of parallel algorithms. In *Advances in computers*. Academic Press, 1980.
[13] F. S. E. Limited. Failures-divergence refinement: FDR2. `http://www.fsel.com/`.
[14] T. Margaria, M. Mendler, and C. Gsottberger. Modelling and verification of unbounded length systolic arrays in monadic second-order logic. In *INFINITY'98*, pages 11–25, 1998.
[15] T. Nipkow, L. C. Paulon, and M. Wenzel. *Isabelle/HOL*. LNCS 2283. Springer, 2002.
[16] L. C. Paulson. *A Generic Theorem Prover*. LNCS 828. Springer, 1994.
[17] N. Petkov. *Systolic Parallel Processing*. North Holland, 1993.
[18] A. W. Roscoe. *The Theory and Practice of Concurrency*. Prentice Hall, 1998.
[19] A. W. Roscoe and N. Dathi. The pursuit of deadlock freedom. *Information and Computation*, 75(3):289–327, 1987.
[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] L. J. Steggles. Verifying an infinite systolic algorithm using third-order algebraic methods. Technical Report 795, Univ. of Newcastle upon Tyne, School of Computing Science, Apr 2003.
[22] R. Woods, A. Cassidy, and J. Gray. VLSI architecture for FPGAs: A case study. In *IEEE Symposium on FPGAs for Custom Computing Machines*, pages 2–9, Los Alamitos, CA, 1996.