

Embedding the Stable Failures Model of CSP in PVS

Kun Wei and James Heather

Department of Computing, University of Surrey, Guildford, Surrey GU2 7XH, UK
{k.wei, j.heather}@surrey.ac.uk

Abstract. We present an embedding of the stable failures model of CSP in the PVS theorem prover. Our work, extending a previous embedding of the traces model of CSP in [6], provides a platform for the formal verification not only of safety specifications, but also of liveness specifications of concurrent systems in theorem provers. Such a platform is particularly good at analyzing infinite-state systems with an arbitrary number of components. We demonstrate the power of this embedding by using it to construct formal proofs that the asymmetric dining philosophers problem with an arbitrary number of philosophers is deterministic and deadlock-free, and that an industrial-scale example, a ‘virtual network’ [21], with any number of dimensions, is deadlock-free. We have established some generic proof tactics for verification of properties of networks with many components. In addition, our technique of integrating FDR and PVS in our demonstration allows for handling of systems that would be difficult or impossible to analyze using either tool on its own.

Keywords: CSP, theorem prover, liveness, deadlock, determinism.

1 Introduction

Concurrent systems are often complex because they consist of many components that can run independently and simultaneously. Proving properties of these systems is also often a difficult task. CSP provides a rich notation for modelling these kinds of system, and the many laws of CSP can be used to verify specifications of such systems, thus enabling designers to check whether the systems meet desired properties or not. However, constructing proofs of correctness by hand is arduous and error-prone.

One highly successful solution to this problem is FDR [9], which is a powerful model-checking tool providing automated analysis and verification of CSP process descriptions. In conjunction with many advanced techniques including data independence [11] and hierarchical compression [15], FDR can in many cases deal efficiently with processes with vast or even infinite state spaces. However, most classes of infinite-state processes are out of reach of model-checking with current techniques. Data independence allows model-checking of systems that have an infinite state space on account of an infinite datatype, but not of systems with an arbitrary number of concurrent processes. The alternative is to take a theorem-proving approach, which allows us to reason about arbitrary processes.

PVS [4, 5], the *Prototype Verification System*, is an interactive theorem prover based on a form of higher-order logic. It provides an environment for constructing precise specifications, and for efficient mechanized verification. Although it is similar in many ways to other theorem provers such as Isabelle/HOL [13] and IMPS [8], it supports a richer type system, and checks semantic consistency for a PVS specification. PVS is also a natural choice for this work because of previous work in [6, 19, 7] where the authors represent the denotational semantics of the traces model of CSP in PVS and then apply their proof strategy to model and verify various safety properties of security protocols. Since the stable failures model records both traces and failures information, we choose to take Dutertre and Schneider's PVS traces embedding and augment it with stable failures.

The extension from Dutertre and Schneider's encoding of the traces model to our embedding of the stable failures model is not at all trivial. They do not consider various important operators of CSP, such as successful termination and sequential composition; our embedding, however, does include these operators, along with various laws about their behaviour. In addition, in order to verify deadlock freedom and determinism, we have formally proved many crucial rules, including the unique fixed point theorem, deterministic induction and various deadlock rules. These rules have previously been proved only by hand; we here give rigorous machine-verified proofs.

Dutertre and Schneider's embedding could prove only safety properties; our platform can verify liveness properties (for example, deadlock freedom and determinism), which cannot be analyzed in the traces model. We will show in this paper how to prove determinism and deadlock freedom of the asymmetric dining philosophers network with an arbitrary number of philosophers, using mathematical induction. In the case of deadlock freedom, the work in PVS essentially reduces the problem to a very small model-checking verification exercise involving under 100 states. Although the proof could be completed entirely in PVS, it would be extremely tedious and time-consuming to perform this model-checking manually in a theorem prover; the more natural approach, and the one adopted here, is to use FDR to complete the finite-state model-checking part of the verification. The main idea of the proof comes from [15], which uses a hierarchical compression technique in FDR to prove the case with very large numbers of philosophers.

Moreover, we have formally proved some of the deadlock rules described in [14], which can be used to construct deadlock-free networks. These formal proofs provide rigorous verification of the rules. The significance of these rules is that FDR can then verify deadlock freedom of complex networks by analysis of individual components of the network. Here we show how to construct the formal proof of these rules, and then use the rules to prove deadlock freedom of a case study in PVS.

In contrast to model-checking, embedding the semantics of CSP into higher-order logics provides mechanical support for verifying the correctness of properties in a system. In the early stage, Camilleri [2] has shown how a theorem prover based on higher-order logic can provide a natural framework for mech-

anizing CSP. However, his mechanization was slightly restricted since both the semantics of CSP and theorem-proving tools have been improved over the past decade.

Tej and Wolff [20] provide a basic platform of encoding the denotational semantics of the CSP failures/divergences model in Isabelle/HOL, along with verifying the consistency of theories and a number of algebraic laws. Our experience suggests, however, that simply providing an embedding is far from sufficient to allow one to verify properties of systems in practice. We therefore have built up a large number of theorems and lemmas to support the verification of particular properties of practical systems. Isobe and Roggenbach [10] propose a new tool called CSP-Prover which provides an encoding of the CSP stable failures model. It appears that this encoding, based on the theorem prover Isabelle/HOL, is essentially an extension of Tej and Wolff’s work; their formalization supports the theory of complete metric spaces as well as the theory of complete partial orders, allowing it to deal with a much wider class of properties of recursion. We have taken a similar approach in our model; furthermore, we have established a class of generic proof tactics, and shown how to combine the use of FDR and a theorem prover so that we are able to model and verify properties of many different types of system.

Brooke [1] uses Timed CSP and PVS and FDR to construct tool-supported proofs to verify properties of systems on an industrial scale. Another successful case is the programming language *Circus* [3, 16], which combines CSP and Z to specify, validate and develop real-time programs. All *Circus* refinement laws are proved using the theorem prover ProofPower-Z.

The remainder of the paper is organized as follows. We will give a brief introduction to the notation of CSP and the denotational semantics of the stable failures model; we then show how to embed this model in PVS; we present some generic proof tactics and our case study, proving using our formalization that the asymmetric dining philosophers with an arbitrary number of philosophers is deadlock-free and deterministic, and that a ‘virtual network’ [21] with any number of dimensions is deadlock-free as a consequence of various deadlock rules; finally, we give conclusions and discuss future work.

2 CSP Notation

CSP is an event-orientated language for describing concurrent systems and their interactions. A system can be considered as a process that might be hierarchically composed of many smaller processes. An individual process can be combined with events or other processes by operators such as prefixing, choice, parallel composition, and so on. There are four semantic models available—traces, stable failures, failures/divergences, and failures/divergences/infinite traces—and which one is chosen depends on what properties of the system one is trying to analyze. In this paper, we choose the CSP stable failures model since this provides a rich enough framework for analysis of deadlock freedom and determinism (for processes known to be non-divergent).

The traces model is the simplest model, in which processes are described according to sequences of events they engage in. The stable failures model, described in detail in [14, 18], records stable failures as well as traces.

Traces tell us exactly what a process can do, but nothing about what it can refuse to do. A refusal set is a set of events from which a process can fail to accept anything no matter how long it is offered; a failure is then defined as a pair (t, X) , where $t \in \text{traces}(P)$ and X is a refusal of the process P after it has performed the trace t . If the trace t can make no internal progress, this failure is called a *stable failure*.

The basic syntax of CSP we use is described by the following grammar:

$$P ::= \text{Div} \mid \text{Stop} \mid \text{Skip} \mid a \rightarrow P \mid P_1 \square P_2 \mid P_1 \sqcap P_2 \mid \\ P_1 \parallel_A P_2 \mid P \setminus A \mid f(P) \mid P_1; P_2$$

where we assume Σ is a universal set including all possible events for processes under consideration, a is an element of Σ , and A is a subset of Σ .

Div is a process which does nothing except diverge. *Stop* is a stable deadlocked process that never performs any events. *Skip* is used to denote successful termination, and it expresses this by means of the termination event \surd , which is not a member of Σ . The process $a \rightarrow P$ behaves like P after performing the event a .

The external choice $P_1 \square P_2$ may behave either like P_1 or like P_2 , depending on what events its environment initially offers. The traces of internal choice $P_1 \sqcap P_2$ are the same as those of $P_1 \square P_2$, but the choice in this case is non-deterministic.

The interface parallel $P_1 \parallel_A P_2$ is the process where all events in the interface A must be synchronized, and other events can be performed independently. The interleaving and alphabetized parallel operators can be defined in terms of interface parallel.

The hiding process $P \setminus A$ will pass through the same events as P , but events in the set A become invisible. The renamed process $f(P)$ means that, for example, an event a in such a process is completely replaced by $f(a)$ where f is a mapping function. The sequential composition $P_1; P_2$ passes control to P_2 when P_1 terminates successfully.

Note that recording only stable failures is not enough because it is not guaranteed that every process has one. For instance, after a process diverges—that is, after it reaches a state from which it can perform an infinite sequence of internal events—it may never reach a stable state, and hence has no more stable failures. Therefore, it is necessary to record traces separately in the stable failures model; each process is represented as the pair $(\text{traces}(P), \text{failures}(P))$.

The stable failures model consists of all those pairs (T, F) with $T \subseteq \Sigma^* \surd$ and $F \subseteq \Sigma^* \surd \times \mathbb{P}(\Sigma^\surd)$ ¹ that satisfy the following conditions:

¹ Σ^* is the set of all finite sequences over Σ and $\mathbb{P}(\Sigma)$ is a powerset; $\Sigma^* \surd = \Sigma^* \cup \{t \surd \mid t \in \Sigma^*\}$ and $\Sigma^\surd = \Sigma \cup \{\surd\}$.

- T is non-empty and prefix closed (SF1)
- $(t, X) \in F \Rightarrow t \in T$ (SF2)
- $(t, X) \in F \wedge Y \subseteq X \Rightarrow (t, Y) \in F$ (SF3)
- $(t, X) \in F \wedge (\forall a \in Y)(t \hat{\ } \langle a \rangle \notin T \Rightarrow (t, X \cup Y) \in F)$ (SF4)
- $t \hat{\ } \langle \surd \rangle \in T \Rightarrow (t, \Sigma) \in F$ (SF5)
- $t \hat{\ } \langle \surd \rangle \in T \Rightarrow (t \hat{\ } \langle \surd \rangle, X) \in F$ (SF6)

The stable failures model deliberately ignores divergence; in situations in which divergence is not an issue, this brings considerable convenience in the form of reduced complexity of the model. For instance, if we know in advance that a process is divergence-free, using the stable failures model can greatly reduce the complexity of the refinement (regardless of whether we are doing theorem-proving or model-checking).

Divergence is not considered as deadlock in the stable failures model, though it is considered as deadlock in failures/divergences. This is precisely what we need here: we shall make considerable use of the fact that hiding of events makes no difference to deadlock freedom. Our formalization follows the denotational semantics of CSP. Detailed semantics of the stable failures model can be found in [14].

3 Embedding CSP Semantics in PVS

As a first step, we need to formalize the CSP notation in PVS. Dutertre and Schneider’s embedding of the traces model in PVS [6] already defines most of the notation that we need; we extend it to the stable failures model, introducing along the way the new operators and laws of CSP that we will require.

The stable failures model is represented by pairs (T, F) in which T is a set of traces that forms the semantics of a process in the traces model. The classic formalization of traces is to simply consider traces as lists of events.

The special event \surd is not a member of Σ and can never be performed by a process unless this is the last event that it engages in. To represent the extended alphabet Σ^\surd , we define a datatype as follows:

```

E [T:TYPE]: DATATYPE WITH SUBTYPES TE, NTE
BEGIN
  tick:tick?:TE
  ES(a:T):non_tick?:NTE
END E

```

where we also define two subtypes TE and NTE. Here, NTE is used to represent Σ .

PVS provides a predefined abstract datatype list. Thus, the type trace defined as follows is simply a subtype of list.

```

trace: TYPE ={ l:list[E] | tick_free?(front(l)) }

```

where the function `front` returns the entire list except for the final element, and `tick_free?` is a predicate that determines whether or not the list includes the event \checkmark . The expression above therefore ensures that \checkmark cannot appear except at the end of a trace.

3.1 Processes

Processes in the stable failures model consist of pairs (T, F) that satisfy the six conditions mentioned in Section 2. Our definition of processes relies on PVS subtyping: `process` is a subtype of `SF` defined as follows:

```
SF: VAR [set[trace[T]], set[[trace[T],set[E]]]]
process:TYPE= {SF | SF1(SF) and SF2(SF) and SF3(SF) and
                SF4(SF) and SF5(SF) and SF6(SF) }
```

where `SF1–SF6` are the six predicate type functions derived from the stable failures model’s conditions (SF1)–(SF6) from Section 2. Note that `T` is a type parameter which denotes the type of elements of a trace, and `trace[T]` will automatically add in the special event \checkmark .

Table 1. CSP syntax

Operation	CSP	CSP _M	PVS
Stop	<i>Stop</i>	STOP	Stop
Skip	<i>Skip</i>	SKIP	Skip
Prefix	$a \rightarrow P$	$a \rightarrow P$	$a \gg P$
External choice	$P_1 \square P_2$	$P_1 \square P_2$	$P_1 \vee P_2$
Internal choice	$P_1 \sqcap P_2$	$P_1 \sqcap P_2$	$P_1 \wedge P_2$
Interface parallel	$P_1 \parallel_A P_2$	$P_1 [A] P_2$	$\text{Par}(A)(P_1, P_2)$
Alphabetized parallel	$P_1 \parallel_A \parallel_B P_2$	$P_1 [A B] P_2$	$\text{Par}(A, B)(P_1, P_2)$
Interleave	$P_1 P_2$	$P_1 P_2$	$P_1 // P_2$
Hiding	$P \setminus A$	$P \setminus A$	P / A
Renaming	$f(P)$	$P[a \leftrightarrow b]$	$\text{Re}(P, f)$
Sequential composition	$P_1; P_2$	$P_1; P_2$	$\text{Seq}(P_1, P_2)$

All of CSP’s main operators are listed in Table 1, with the standard CSP syntax, the CSP_M syntax (as used in FDR), and PVS’s syntax. Note that in this paper, we consider only injective renaming since it leaves the behaviour of a process unchanged except for the names of the actions, and it thus has a rich set of laws. Even so, injectivity is not sufficient for some laws in the stable failures model: sometimes we need the renaming function to be bijective. (This is clearly an issue only when Σ is infinite.)

We also define *indexed* versions of the choice and parallel operators, which are often used in analyzing a large network. In particular, we use $\text{Echoice}(P)$ and $\text{Par}(A)(P)$ to denote $\square_{i \in I} P_i$ and $\parallel_{i=1}^n (P_i, A_i)$ respectively, where P is a parametric process and A is a parametric set.

FDR’s main function is to determine whether one process refines another. In the stable failures model, this equates to checking whether the traces and failures of one process are subsets of the traces and failures of the other:

$$P \sqsubseteq_F Q \equiv \text{traces}(P) \supseteq \text{traces}(Q) \wedge \text{failures}(P) \supseteq \text{failures}(Q)$$

The idea of refinement is still kept in verifying properties of processes in PVS. For example, for proving a process Q deadlock-free, we often explicitly construct a deadlock-free specification P , then check whether Q refines P or whether Q is a subset of P . Obviously, if Q refines P and P is deadlock-free, then Q is deadlock-free as well.

We use the relation ‘ \leq ’ to denote refinement of processes in PVS: $Q \leq P$, representing $P \sqsubseteq Q$ in CSP, corresponds to $Q \subseteq P$. Since ‘ \leq ’ and subset? have been predefined in the prelude library of PVS, we rewrite them so that they can compare a pair of sets. So \leq is defined as the following:

```

 $\leq(Q, P) : \text{bool} = \text{subset?}(Q, P)$ 
```

3.2 Fixed Points and Recursive Processes

Some processes, called recursive processes, may run indefinitely, instead of executing for a finite number of steps and then stopping; Unfortunately, we cannot define such processes directly in PVS since a theorem prover will not allow us to get away with any kind of recursive definition unless we can demonstrate that it is well-defined.

The formalization used in [6] to deal with recursive processes is the ‘ μ -calculus’ theory, which uses a μ operator (‘ mu ’ in PVS) to compute the least fixed point of a monotonic function². We have extended this to the stable failures model since all CSP operators are monotonic over the stable failures model with respect to the refinement order and the subset order. We also have proved a general fixed point induction theorem, which is crucial in analyzing refinement of recursive processes:

```

induction: PROPOSITION
(FORALL X : X<=H IMPLIES F(X)<=H) IMPLIES mu(F)<=H
```

We also have extended the least-fixed-point theory to represent mutually recursive processes. The general case of a mutual recursion is concerned with a family or *vector* of processes \underline{X} , and the recursive definition then takes the

² A monotonic function in this context is a function F such that if $Q \leq P$ then $F(Q) \leq F(P)$.

form $\underline{X} = \underline{F}(\underline{X})$ where \underline{F} is a function from a vector of processes to a vector of processes. It is still appropriate to use the least fixed point of the function \underline{F} to represent a mutual recursion. In addition, we have proved that all lemmas and induction theorems of the least fixed point still hold in mutual recursions.

In order for fixed points to be useful, we will usually want to show that a function has a unique fixed point. Roscoe [14] shows how to apply a restriction operator and a constructive function to demonstrate the existence of a unique fixed point. We first formally define the restriction operator

$$\text{chop}(P, n) : \text{process}[T] = (\{t \mid P'1(t) \text{ and } \text{length}(t) \leq n\}, \\ \{(t, A) \mid P'2(t, A) \text{ and } \text{length}(t) < n\})$$

where the purpose of ‘*chop*’ is to restrict the process P so that it can never perform any traces of greater than length $n \in \mathbb{N}$. Note that we here use ‘ $<$ ’ for stable failures in the definition because we want to make such a definition consistent with a fact that Div is the least element in the subset order.

We then say that F is *constructive* if

$$\text{constructive?}(F) : \text{bool} = \text{FORALL } P, Q, n : \text{chop}(P, n) = \text{chop}(Q, n) \\ \text{IMPLIES } \text{chop}(F(P), n+1) = \text{chop}(F(Q), n+1)$$

For a function F , we have that whenever $F(X) = X$ and $F(Y) = Y$ then $X = Y$, then we say that F has a unique fixed point. The mathematical background of the unique fixed point theorem is not covered in this paper; Roscoe [14] gives a detailed explanation in terms of partial orders and of metric spaces.

In addition, we have proven a number of algebraic laws which are essential in the verification of properties of processes, whereas these laws can help us to verify the consistency of the CSP semantics.

4 Generic Proof Tactics

Our aim in embedding the denotational semantics of the stable failures model of CSP into PVS is not only to verify the consistency of theories and algebraic laws of CSP, but also to build up some strategies so that we can check properties of various infinite-state systems. The focus is especially on liveness properties, which cannot be analyzed in the traces model. Our first step in this direction is the verification of some general properties such as determinism and deadlock freedom.

4.1 Determinism

A deterministic process always behaves in the same way when offered exactly the same inputs. The most obvious practical benefit is that this kind of process is testable because its behaviour does not vary unless the external inputs are changed.

Of course, only processes known to be divergence-free can be verified in the stable failures model, because this model cannot detect divergence. In Figure 1,

```

determinism [T:TYPE ] : THEORY
BEGIN
  IMPORTING fixed_points[T]

  t: VAR trace[E]
  a: VAR E
  n: VAR nat
  A,B: VAR set[E]
  P,Q,X: VAR process[E]
  F: VAR [process[E]->process[E]]

  DET?(P):bool= FORALL t,a: P'1(add(t,a))
                IMPLIES NOT P'2((t,singleton(a)))

  det_stop: LEMMA DET?(Stop[E])
  det_prefix: LEMMA DET?(P) IMPLIES DET?(a>>P)
  det_par: LEMMA DET?(P) AND DET?(Q) IMPLIES DET?(Par(A,B)(P,Q))

  det_seq: LEMMA DET?(P) AND DET?(Q) IMPLIES DET?(Seq(P,Q))
  det_chop: LEMMA DET?(P) IFF (FORALL n: DET?(chop(P,n)))
  det_subset: LEMMA (DET?(P) AND Q <= P ) IMPLIES DET?(Q)

  det_induction: LEMMA ( constructive?(F) AND (EXISTS X: DET?(X))
                        AND (FORALL X: DET?(X) IMPLIES DET?(F(X))) )
                  IMPLIES DET?(mu(F))

END determinism

```

Fig. 1. Examples of deterministic rules

the definition DET? states that a deterministic process can not accept an event \mathbf{a} as well as being able to refuse this event; here, $\text{add}(\mathbf{t}, \mathbf{a})$ adds the event \mathbf{a} onto the end of the trace \mathbf{t} . Note that \mathbf{E} has been previously defined as a datatype including the special event \checkmark .

Some CSP operators preserve determinism: if P and Q are deterministic then so are Stop , $a \rightarrow P$, $P \parallel_B Q$ and sequential composition $P;Q$. Such laws and some useful lemmas are also listed in Figure 1. Furthermore, if $\text{initials}(P)$ and $\text{initials}(Q)$ are disjoint then $P \square Q$ is also deterministic. Here, $\text{initials}(P)$ is the set of all of P 's initial events; for example, it can be defined as follows:

$$\text{initials}(P) = \{a \in \Sigma^{\checkmark} \mid \langle a \rangle \in \text{traces}(P)\}$$

Proving determinism of non-recursive processes is often not difficult but it can be time consuming. For recursive processes, one has to apply an induction rule such as det_induction in Figure 1 to make any progress; this rule states

that if F is constructive and determinism-preserving then the least fixed point of F is also deterministic.

Note that the induction rule here does not imply that every recursive deterministic process is the least fixed point of a constructive determinism-preserving function. In addition, it is also possible in some cases to infer the determinism of $mu(F)$ directly. Usually, however, the easiest way to prove that a recursive process is deterministic is by means of this theorem. For this reason, the determinism induction theorem proved here will be extremely useful in many applications.

4.2 Deadlock Freedom

One of the most important concepts concerning concurrent systems is deadlock, which arises when no further progress can be made. Deadlock is a kind of liveness property, so we cannot detect or reason about it using traces alone. The stable failures model, however, is quite suitable for describing deadlock freedom. The definition of deadlock freedom as well as some laws are given in Figure 2.

Divergence is considered deadlock-free in the stable failures model, while it is not deadlock-free in the failures/divergences model. The usual way to prove deadlock freedom of a recursive process is to define a deadlock-free specification explicitly, and prove that the process is a refinement of the specification; then obviously the refining process is deadlock-free as well.

```

deadlock_free [T: TYPE] : THEORY
BEGIN
  ....
  a: VAR E
  t: VAR trace[E]
  P,Q: VAR process[E]
  A: VAR set[E]
  f: VAR [set[E]->set[E]]

  DLF?(P):bool = FORALL t: P'1(t) IMPLIES NOT P'2((t,fullset))

  dlf_prefix: LEMMA DLF?(P) IMPLIES DLF?(a>>P)
  dlf_echoice: LEMMA DLF?(P) AND DLF?(Q) IMPLIES DLF?(P\Q)
  dlf_hide: LEMMA DLF?(P) IFF DLF?( P / A )
  dlf_rename: LEMMA injective?(f)
                    IMPLIES (DLF?(P) IFF DLF?(Re(P,f)))
  dlf_subset: LEMMA subset?(P,Q) AND DLF?(Q) IMPLIES DLF?(P)
  ....
END deadlock_free

```

Fig. 2. Generic deadlock-free rules

Figure 2 also shows two important laws, `dlf_hide` and `dlf_rename`, that are extremely useful in the analysis of deadlock freedom in the stable failures model. These two facts underpin the definition of deadlock freedom: deadlock means reaching a state where no further progress is possible regardless of whether the actions are renamed or hidden.

Deadlock freedom is a global property; in other words, we cannot guarantee that if all components of a network are individually deadlock-free then the whole network will also be deadlock-free. Often, the complexity and the work of verification of a particular property can be greatly reduced by decomposing a global property of a network into local properties of the network’s components; this is not easy to do, however, with deadlock freedom.

There are, however, some deadlock rules that can be used to analyze a large network locally rather than considering the whole network all the time. Roscoe [14] gives various deadlock rules, and shows how to apply these rules to prove deadlock freedom of some large networks. We have proved some of these deadlock rules at a formal level, in order to be able to construct formal proofs of deadlock freedom of various networks.

The terminology introduced here is taken from [14]. We consider a network $V = \parallel_{i=1}^n (P_i, A_i)$, which is a parallel composition of a finite sequence of processes $\langle P_1, \dots, P_n \rangle$ and their alphabets. We shall suppose that V is triple-disjoint³, and that no component process ever terminates or deadlocks. In such a network, a state is defined as the pair $(s, \langle X_1, \dots, X_n \rangle)$ in a network V where $s \in (\bigcup_{i=1}^n A_i)^*$, $(s \upharpoonright A_i, X_i) \in failures(P_i)$, and $X_i \supseteq \Sigma \setminus initials(P_i / (s \upharpoonright A_i))$. Here, \upharpoonright is the projection operator and \setminus is to calculate difference of two sets. Therefore, a state is in deadlock if the union of all refusal sets X_i is equal to the Σ .

The concepts that we shall need, such as *ungranted request*, *conflict* and so on are now straightforward to define formally in Figure 3. Note that we here completely ignore the event \checkmark since the assumption is that no P_i can terminate. In a state $(s, \langle X_P, X_Q \rangle)$, we say there is an *ungranted request* from P to Q in the composition $P \parallel_A \parallel_B Q$ if P can communicate in B but they can not agree on any communication in $A \cap B$. Obviously, ungranted requests are the underlying factors that result in deadlock. We here use a predicate `ung_request?(A,B)(P,Q)(t,X1,X2)` in Figure 3 to define such an ungranted request.

There is a *conflict* between P and Q if there is an ungranted request in both directions. The formal definition may be expressed as `CF?(A,B)(P,Q)` in Figure 3. Additionally, a *strong conflict* is a conflict in which one of the two processes has its only ungranted request to the other. Finally, a network V is *conflict-free* if no pair of its nodes is in conflict. We here use `CFF?(X)(S)` to describe this property in Figure 3.

The following fundamental result quoted from [14] underlies all of the deadlock rules.

³ If P_i, P_j and P_k are three distinct nodes of V , then $A_i \cap A_j \cap A_k = \emptyset$.

```

conflict [T:TYPE]: THEORY
  BEGIN
  ....
  A,B,X1,X2:VAR set [T]
  P,Q: VAR process [T]
  t: VAT trace [T]
  ....
  ung_request?(A,B)(P,Q)(t,X1,X2):bool = P'2(proj(t,A),X1) AND
    Q'2(proj(t,B),X2) AND subset?(sigma(t),union(A,B)) AND
    subset?(complement(initials(P,proj(t,A))),X1) AND
    subset?(complement(initials(Q,proj(t,B))),X2) AND
  subset?(union(complement(X1),complement(X2)), intersection(A,B))
    AND intersection(B,complement(X1)) /= emptyset AND
    subset?(intersection(B,complement(X1)),X2)

  CF?(A,B)(P,Q):bool=EXISTS t,X1,X2: ung_request?(A,B)(P,Q)(t,X1,X2)
    AND ung_request?(B,A)(Q,P)(t,X2,X1)

  SCF?(A,B)(P,Q):bool=EXISTS t,X1,X2: ung_request?(A,B)(P,Q)(t,X1,X2)
    AND ung_request?(B,A)(Q,P)(t,X2,X1)
    AND ( subset?(complement(X1), B) OR subset?(complement(X2), A) )
  ....
  CFF?(X)(S):bool= FORALL i,j: i/=j IMPLIES
    NOT CF?(X(i),X(j))(S(i),S(j))

  SCFF?(X)(S):bool= FORALL i,j: i/=j IMPLIES
    NOT SCF?(X(i),X(j))(S(i),S(j))
  ....
  END conflict

```

Fig. 3. The definitions of ungranted request and conflict

Fundamental Principle of Deadlock. *If V is a network which satisfies our basic assumptions and which is free of strong conflict, then any deadlock state of V contains a proper cycle of ungranted requests.*

To prove this law, we have to define a finite network, as in Figure 4, which guarantees the existence of a proper cycle of ungranted requests. One of the most important laws with regard to ungranted requests is `dl_ung_request` which shows that in a deadlock state, for any node, there always exist two other distinct nodes such that the three nodes together form a sequence of ungranted requests.

The proof of this fundamental law comes as a fairly straightforward consequence of the lemma `dl_ung_request`, because any ungranted request from any process S_i to another process S_j will then guarantee an ungranted request from S_j to some S_k , and so on; this sequence must repeat since the network is finite. In the formal definition listed in Figure 4, we use a predicate `DL?(V'2)(V'3)`

```

deadlock_rules[T:TYPE]: THEORY
BEGIN
  ....
  NET:TYPE=[size, [below[size]->set[E]], [below[size]->process[T]]]
  V: VAR NET

  dl_ung_request: LEMMA (Assump?(X)(S) AND SCFF?(X)(S) AND DL?(X)(S))
                    IMPLIES
                    FORALL i:EXISTS j,k: (S(i)/=S(j) AND S(j)/=S(k) AND S(i)/=S(k))
                    AND (EXISTS t,X1,X2:ung_request?(X(i),X(j))(S(i),S(j)(t,X1,X2)))
                    AND (EXISTS t,X1,X2:ung_request?(X(j),X(k))(S(j),S(k)(t,X1,X2)))

  ....
  fundamental_principle: LEMMA (ASSUMP?(V'2)(V'3) AND CFF?(V'2)(V'3)
                                AND DL?(V'2)(V'3)) IMPLIES cycle?(V)

  pre_rule2?(V):bool = ASSUMP?(V'2)(V'3) AND CFF?(V'2)(V'3) AND
    ( (EXISTS t,X1,X2: V'3(j)<=V'3(i) AND
      ung_request?(V'2(i),V'2(j))(V'3(i),V'3(j))(t,X1,x2))
      IMPLIES ( FORALL (k:{x:nat|V'3(x)<=V'3(i) AND
        intersection(V'2(x),V'2(i))/=emptyset}):
        EXISTS t,X1,X2:
          ung_request?(V'2(k),V'2(i))(V'3(k),V'3(i))(t,X1,X2) ) )

  deadlock_rule2: LEMMA pre_rule2?(V) IMPLIES (NOT DL?(V'2)(V'3))
  ....
END deadlock_rules

```

Fig. 4. The deadlock rules

to denote deadlock of the network, and use a predicate $\text{cycle?}(V)$ to show that there exists at least one cycle of ungranted requests in the network.

By making use of this fundamental principle, we have proved Deadlock Rule 2 quoted from [14] as well:

Deadlock Rule 2. *Suppose V is conflict-free and has a node ordering $<$ such that whenever node P_i has a request to any P_j with $P_j < P_i$, then it has a request to all its neighbours P_k such that $P_k < P_i$. Then V is deadlock free.*

The formal proof just translates the one given in [14] into PVS. If V can deadlock, then there is a cycle of ungranted requests which must contain one maximal P_i ; necessarily P_i has an ungranted request to P_{i+1} less than itself, then it also has a request to P_{i-1} ; and this violates the assumption of conflict freedom. Such a rule is formally expressed in Figure 4 where \leq denotes a partial order, and we also find out any process's neighbours by only comparing their algebras.

5 Case Study

We show the power of the formalization of CSP semantics by two examples: the dining philosophers problem and the ‘virtual network’ [21].

5.1 The Dining Philosophers Problem

The dining philosophers problem was first described by Edsger W. Dijkstra in 1965. It is a classic multi-process synchronization problem. The problem consists of n philosophers sitting at a table with a bowl of spaghetti in the middle. Between each pair of adjacent philosophers, there is a single fork; and to eat, a philosopher must be holding both of the forks that are beside him. We assume all philosophers pick forks up in the same order—right hand first—and do not put down any fork they have picked up until they have grabbed both. Figure 5 shows the dining philosophers network’s structure, composed of philosopher/fork pairs.

It is quite straightforward to prove determinism of the n dining philosophers problem in combination with the `det_induction` rule in Figure 1 and the properties of various CSP operators. In Figure 6, $H(i, j)(X)$ and $F(i, j)(X)$ are used to express the behaviour of an individual philosopher and fork respectively where i denotes the total number of philosophers; `pick(j, j)` denotes that the j th philosopher picks up the j th fork, and so does `putdown(j, j)`; `inc(i, j)` denotes addition modulo i . Note that each philosopher and fork process is parameterized not only by its index but also by the total number of philosophers, since this affects the modular calculation. Moreover both the philosopher and the fork are recursive processes, and we use the least fixed points of the functions H and F to represent them in PVS. Here $\text{PandF}(i, j)$ is used to represent the combination of

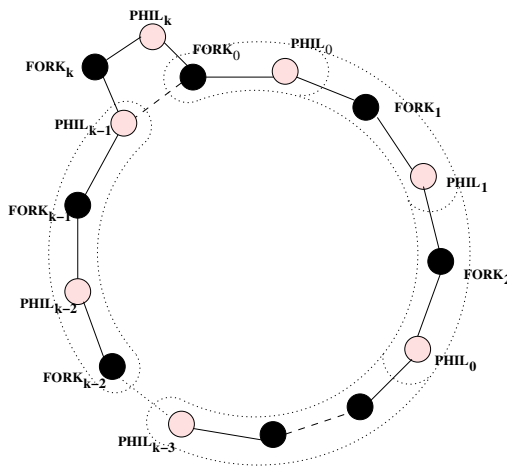


Fig. 5. Inductive structure of dining philosophers

```

philosopher_det: THEORY
BEGIN
  ....
  H(i,j)(X): process[events] = pickup(j,j)>>(pickup(j,inc(i,j))>>
    (putdown(j,inc(i,j))>>(putdown(j,j)>>X)))
  F(i,j)(X): process[events] =( (pickup(j,j)>>(putdown(j,j)>>X))
    \ / (pickup(dec(i,j),j)>>(putdown(dec(i,j),j)>>X)))

  PHIL(i,j): process[events] = mu(H(i,j))
  FORK(i,j): process[events] = mu(F(i,j))
  PandF(i,j):process[events] =
    Par(AP(i,j),AF(i,j))(PHIL(i,j),FORK(i,j))
  ....

  P(n)(m):process[events] = PandF(n,m)
  APF(n)(m):set[events] = union(AP(n,m),AF(n,m))
  COLLEGE(n): process[events] = Par(APF(n))(P(n))

  fork_det: LEMMA DET?(FORK(i,j))
  phil_det: LEMMA DET?(PHIL(i,j))
  pair_det: LEMMA DET?(PandF(i,j))
  college_det: LEMMA DET?(COLLEGE(n))
  ....
END philosopher_det

```

Fig. 6. proving determinism of the dining philosophers problem

a philosopher and his right-hand fork where $AP(i,j)$ and $AF(i,j)$ denote their alphabets.

For constructing the proof, we need only to prove that the processes $PHIL(i,j)$ and $FORK(i,j)$ are deterministic; then the alphabetized parallel combination $PandF(i,j)$ is deterministic by means of the `det_par` rule in Figure 1; the entire system $COLLEGE(n)$ is then also deterministic since it consists of $PandF(n,m)$ for $m < n$.

Deadlock freedom is a more tricky issue. Obviously for the dining philosophers problem, the one and only one situation causing deadlock is that in which all philosophers hold their right-hand forks simultaneously and wait for their neighbours to put down their forks. There are many modifications one can make to avoid deadlock, one of which results in the asymmetric dining philosophers problem: one philosopher picks up a left-hand fork first.

The basic strategy we adopt is similar to an induction used in [15], where the authors use a hierarchical compression technique in FDR to prove the case with huge numbers of philosophers. The key idea is that by hiding their internal events and carefully renaming their interface events, we can prove that any number ($n > 1$) of right-handed pairs of philosophers and forks are equivalent. The

```

philosopher_dlf:THEORY
BEGIN

....
H(i,j)(X): process[events] =
  IF j=0 THEN pickup(j,inc(i,j))>>(pickup(j,j)>>
    (putdown(j,j)>>(putdown(j,inc(i,j))>>X)))
  ELSE pickup(j,j)>>(pickup(j,inc(i,j))>>
    (putdown(j,inc(i,j))>>(putdown(j,j)>>X)))
  ENDIF
....

PL(n:{x:int|x>2})(m:{x:int|m>0 and m<n})
  :process[events]= PandF(n,m)
C(n):process[events] = Par(APF(n))(PL(n))
COLLEGE(n):process[events] = Par(I(n))(PandF(n,0),C(n))
....

phil3_dlf: LEMMA DLF?(COLLEGE(3))
phil_key: ASSUMPTION
  (Par(APF2(k))(PL2(k))/IE(k)
   =Re((Par(APF3(k+1))(PL3(k+1))/IE(k+1)), f)

phil_dlf_hr: LEMMA C(n)/IE(n) = Re(C(n+1)/IE(n+1),f)
phil_dlf: LEMMA DLF?(COLLEGE(n))
....
END philosopher_dlf

```

Fig. 7. Proving the asymmetric dining philosophers problem deadlock-free

proof starts from the case with $n = 3$ philosophers; then, for the inductive step, we assume that the case of $n = k$ philosophers is deadlock-free, and show that the system remains deadlock-free when the number of philosophers is $n = k + 1$.

Figure 7 roughly shows the inductive steps of proving that the asymmetric dining philosophers network is deadlock-free. First of all, the definition of the philosophers has been changed since we force the zeroth philosopher to pick up his left-hand fork first. Figure 5 also shows how we deduce deadlock freedom of $k + 1$ philosophers from the case of k philosophers. The key to achieving this step is to prove the equivalence of two processes: k philosophers and $k + 1$ philosophers. Such an idea is proved in the lemma `phil_dlf_hr` in Figure 7 where $C(n)$ is restricted to be the parallel combination of pairs of philosophers and forks without involving the pair of the zeroth philosopher and his right-hand fork.

Certainly, it is unnecessary to compare all pairs, and we need to concentrate only on the last two pairs in the circle of Figure 5. The key to the induction is

that if we hide the internal events of the parallel combination of $\text{PandF}(k, k-2)$ and $\text{PandF}(k, k-1)$, it is equivalent to the parallel combination of $\text{PandF}(k+1, k-2)$, $\text{PandF}(k+1, k-1)$ and $\text{PandF}(k+1, k)$ with their internal events hidden and $\text{pickup}(k, 0)$ and $\text{putdown}(k, 0)$ renamed as $\text{pickup}(k-1, 0)$ and $\text{putdown}(k-1, 0)$ respectively. Therefore, it is transformed into the lemma `phil_key` in Figure 7 in which $\text{IE}(k)$ and $\text{IE}(k+1)$ denote the sets of internal events and f is a bijective function which performs the renaming operation. To get the final result that the case of $k + 1$ philosophers is deadlock-free, we have to combine two laws, `dlf_hide` and `dlf_rename`, which are mentioned in the above section. Consequently, the proof is completely established in the lemma `phil_dlf` in Figure 7.

Note that although it would be possible to prove the lemma `phil_key` in PVS, it would be in one sense perverse to do so, since it is essentially a very small model-checking exercise. It would take a long time to trace through the states of each side one by one checking for correspondence; FDR, on the other hand, can verify the equation in a fraction of a second. The approach we take, therefore, is to build this equation into the PVS theory as an assumption, and then prove it in FDR. In this way, we harness the power of the theorem prover for establishing results about an infinite-state system, whilst retaining the speed and automation of a model-checker for certain small parts of the proof. Using PVS in combination with FDR, then, we have successfully proven the asymmetric dining philosophers network with an arbitrary number of philosophers to be deadlock-free. This strategy of using a theorem prover and a model checker in concert is extremely powerful: the different types of tool complement each other very well. By using both together, we can analyze systems that would be out of reach of either individually.

5.2 The Virtual Network

We now demonstrate the use of Deadlock Rule 2 by means of a routing algorithm example called the ‘virtual network’, quoted in [14] and originally given in [21]. Suppose we want to send a package from any one of the nodes $N_{i,j}$ to any other in a rectangular grid. It seems that the above rule can not directly applied to this system. Roscoe however wisely divides each node $N_{i,j}$ in the system into two parallel processes $I_{i,j}$ and $O_{i,j}$, and defines a partial order such that $I_{i,j} \leq I_{i',j'}$ iff $i \leq i' \wedge j \leq j'$, $O_{i,j} \leq O_{i',j'}$ iff $i \geq i' \wedge j \geq j'$, and $I_{i,j} \leq O_{i',j'}$ for all i, j, i', j' , to satisfy the assumptions of this rule.

This partial order implies that a package is transmitted through $I_{i,j}$ in increasing index order, whereas through $O_{i,j}$ it is in decreasing index order. For example, if a package is sent from $N_{1,3}$ to $N_{2,2}$, then the path is $\langle I_{1,3}, I_{2,3}, O_{2,3}, O_{2,2} \rangle$. The CSP code used to represent such a system using mutual recursion can be found in [14], and we here transform it into PVS in Figure 8 where `IN(i, j)` and `OUT(i, j)` are used to represent the two synchronized processes, and `VN` denotes the entire system. Obviously, this system transparently satisfies the requirements of Rule 2.

```

virtual_network:THEORY
....
F(i,j)(X)(0):process[events] =
  in(x,y,m)>>X(1)\/(I_up(x,y,m)>>X(1) \ I_left(x,y,m)>>X(1))
F(i,j)(X)(1):process[events] =
  IF i<x THEN I_right(x,y,m) >> X(0)
  ELSIF j<y THEN I_down(x,y,m) >> X(0)
  ELSE over(x,y,m) >> X(0) ENDIF

H(i,j)(Y)(0):process[events]=
  over(x,y,m)>>Y(1)\/(O_down(x,y,m)>>Y(1)\O_right(x,y,m)>>Y(1))
H(i,j)(Y)(1):process[events]=
  IF i>x THEN O_left(x,y,m) >> Y(0)
  ELSIF j>y THEN O_up(x,y,m) >> Y(0)
  ELSE out(x,y,m) >> Y(0) ENDIF

IN(i,j):process[events] = mu(F(i,j))
OUT(i,j):process[events] = mu(H(i,j))
....
deadlock_check: LEMMA pre_rule2?(VN)

END virtual_network

```

Fig. 8. The virtual network

By making use of Deadlock Rule 2, we have additionally formally proved that a network with any number of dimensions is deadlock-free. In the definition of such a rule, we use an interpreted type `size` to denote the size of the network; in other words, the number of dimensions of the network can be anything drawn from the type of `size`.

Proving the new network to be deadlock-free needs careful work, because there are a number of issues involved—for instance, checking whether the network meets freedom of conflict, one of the assumptions of Rule 2, and proving that two mutually recursive processes are conflict-free. Along the way, we have constructed various theorems such as the *conflict free induction* theorem to cope with recursive processes. The final result is a proof of correctness that cannot be easily established in a model checker.

6 Conclusion and Future Work

In this paper, we have presented an embedding of the stable failures model of CSP into PVS that preserves the algebraic properties of CSP, and then used this formalism to prove determinism and deadlock freedom of the asymmetric dining philosophers problem with an arbitrary number of philosophers, and an example

of layered routing. Theorem proving is a good complement of model-checking tools such as FDR, which can efficiently verify finite-state systems, but which cannot verify infinite-state systems without outside help.

One of the biggest advantages of a theorem prover is that it is possible to reason about systems with massive or infinite state spaces, admittedly at the cost of sacrificing automatic proof. Verifying a system like our example requires considerable work. However, PVS is a deductive system in which all completed proofs can be used in later proofs. In the course of constructing this proof, we have amassed many lemmas and theorems that will make proving properties of other systems substantially less time-consuming, both for us and for others.

The stable failures model, as well as allowing one to verify properties relating to deadlock freedom, contains sufficient detail to specify many other liveness properties. We are in the process of building up a general platform that provides mechanical assistance for formal analysis of liveness properties of systems. We believe that our model can be used in many different application areas, such as verification of security protocols and general communication protocols. For example, Schneider [17] has modelled and analyzed some properties of a non-repudiation protocol using the traces model of CSP, but some of the other (alleged) properties of the protocol can be formulated only in terms of liveness, and treatment of them requires consideration of failures as well as traces. In addition, we have analyzed and verified the fairness property of the timed Zhou-Gollmann non-repudiation protocol using FDR, and work is in progress on extending this analysis in PVS to cover an infinite network of communicating agents. Denial of service is also naturally specified as a liveness property [12], and one that we expect to be able to use our work to analyze.

We aim in future work to apply our model to other types of network and investigate possible ways to analyze liveness properties of other systems. In our long-term plan, we hope to extend our model to the failures/divergences model; we would like then to extend it further to include infinite traces, which is an area that currently has no tool support at all.

References

1. P. Brooke. *A Timed Semantics for a Hierarchical Design Notation*. PhD thesis, University of York, 1999.
2. A. J. Camilleri. Higher order logic mechanization of the CSP failure-divergence semantics. Technical report, HP Lab Bristol, 1990.
3. A. L. C. Cavalcanti, A. C. A. Sampaio, and J. C. P. Woodcock. A Refinement Strategy for Circus. *Formal Aspects of Computing*, 15(2-3):146–181, unknown 2003.
4. J. Crow, S. Owre, J. Rushby, and N. Shankar. A tutorial introduction to PVS. In *Workshop on Industrial-Strength Formal Specification Techniques*, Boca Raton, Florida, Apr. 1995.
5. J. Crow, S. Owre, J. Rushby, and N. Shankar. *PVS Prover Guide, PVS Language Reference, PVS System Guide*. SRI International, 2001.

6. B. Dutertre and S. A. Schneider. Embedding CSP in PVS: an application to authentication protocols. In E. Gunter and A. Felty, editors, *Theorem Proving in Higher-Order Logics: 10th International Conference, TPHOLs '97*, volume 1275 of *Lecture Notes in Computer Science*. Springer-Verlag, 1997.
7. N. Evans and S. A. Schneider. Analysing Time Dependent Security Properties in CSP using PVS. In *ESORICS 2000*, volume 1895 of *Lecture Notes in Computer Science*. Springer-Verlag, 2000.
8. W. M. Farmer, J. D. Guttman, and J. F. Thayer. IMPS: An Interactive Mathematical Proof System. *Journal of Automated Reasoning*, 11:213–218, 1993.
9. Formal Systems (Europe) Ltd. Failures-Divergence Refinement—FDR 2 user manual, 1997. Available from Formal Systems' web site at <http://www.formal.demon.co.uk/FDR2.html>.
10. Y. Isobe and M. Roggenbach. A generic theorem prover of CSP refinement. *TACAS 2005*, LNCS 3440, 2005.
11. R. Lazić. *A semantic study of data-independence with application to the mechanical verification of concurrent systems*. PhD thesis, Oxford University, 1999.
12. C. A. Meadows. Open issues in formal methods for cryptographic protocol analysis. In V. I. Gorodetski, V. A. Skormin, and L. J. Popyack, editors, *MMM-ACMS*, volume 2052 of *Lecture Notes in Computer Science*, pages 237–250. Springer, 2001.
13. L. C. Paulson. A formulation of the simple theory of types (for isabelle). In P. Martin-Löf and G. Mints, editors, *Conference on Computer Logic*, volume 417 of *Lecture Notes in Computer Science*, pages 246–274. Springer, 1988.
14. A. W. Roscoe. *The Theory and Practice of Concurrency*. Prentice-Hall International, 1998.
15. A. W. Roscoe, P. H. B. Gardiner, M. Goldsmith, J. R. Hulance, D. M. Jackson, and J. B. Scattergood. Hierarchical compression for model-checking csp or how to check 10^{20} dining philosophers for deadlock. In E. Brinksma, R. Cleaveland, K. G. Larsen, T. Margaria, and B. Steffen, editors, *TACAS*, volume 1019 of *Lecture Notes in Computer Science*, pages 133–152. Springer, 1995.
16. A. Sampaio, J. Woodcock, and A. Cavalcanti. Refinement in Circus. In L. Eriksson and P. Lindsay, editors, *FME 2002: Formal Methods - Getting IT Right*, volume 2391 of *Lecture Notes in Computer Science*, pages 451–470. Springer-Verlag, unknown 2002.
17. S. A. Schneider. Formal analysis of a non-repudiation protocol. In *Proceedings of the 11th IEEE Computer Security Foundations Workshop*, 1998.
18. S. A. Schneider. *Concurrent and real-time systems: the CSP approach*. John Wiley & Sons, 1999.
19. S. A. Schneider and J. Bryans. CSP, PVS and a Recursive Authentication Protocol. In *DIMACS Workshop on Formal Verification of Security Protocols*, Sept. 1997.
20. H. Tej and B. Wolff. A corrected failure-divergence model for CSP in Isabelle/HOL. In J. S. Fitzgerald, C. B. Jones, and P. Lucas, editors, *FME*, volume 1313 of *Lecture Notes in Computer Science*. Springer, 1997.
21. J. Yantchev and C. Jesshope. Adaptive, low latency, deadlock-free packet routing for networks of processors. In *IEE Pro E*, May 1989.