Architectures, Languages and Patterns
P.H. Welch and A.W.P. Bakkers (Ed.)
IOS Press, 1998

A Technique for Checking the CSP sat Property

Jeremy M.R. MARTIN
Oxford University Computing Services,
13 Banbury Road,
Oxford 0X2 6NN, UK

Sabah A. JASSIM
Department of Mathematics,
Statistics, and Computer Science,
University of Buckingham,
MK18 1EG, UK

Abstract. This paper presents an algorithm for checking that a CSP process satisfies a specification defined by a boolean-valued function on its traces and refusals, i.e.

$$P\,\mathbf{sat}\,f(tr, ref)$$

This is contrasted with the refinement approach, as implemented by the FDR tool, of checking that one CSP process is a possible implementation of another, i.e

$$P \sqsupseteq \text{SPEC}$$

1 Introduction

The CSP Language of C.A.R.Hoare[3, 8] is a notation for describing patterns of communication by algebraic expressions. It is widely used for the design of parallel and distributed hardware and software, and for the formal proof of vital properties of such systems. However, without computer assistance, it is often impractical to prove such properties other than for toy systems.

There are two standard approaches to specifying properties of a CSP process *P*. The first is to use logical '**sat**' clauses to define constraints on the directly observable behaviour patterns of *P*. The second is to provide an abstract non-deterministic process which characterises these constraints and which must be *refined* by *P*. Both these methods are described in section 2. Automated support for the latter technique of refinement checking is given by the FDR tool of Formal Systems Europe Ltd.[2] – this paper describes a new algorithm for checking automatically the former technique of specifying behaviour using **sat** clauses.

The rest of this paper is structured as follows. In section 2 we review the CSP language and its two aforementioned specification techniques. In section 3 we describe the *normal-form transition system* which was developed for use with FDR and is also crucial to the new algorithm introduced in section 4. The other vital ingredient for our new technique is the *incremental trace function*, which enables us to prove properties of infinite sets of behaviour patterns through finite analysis. Some useful examples of such functions are given in section 5. In section 6 we provide two case studies to demonstrate the practical significance of our new technique and we finish off with a discussion of further applications of our new method and directions for future research.

Two appendices are included. The first lists some relevant CSP terminology for specifying properties of traces. The second outlines the relationship between the occam programming language and CSP.

2 The CSP Language, Specification and Refinement

The core syntax of CSP is described by the following grammar

Process $::==$	$\mathit{STOP} \;\mid$	Deadlock
	$\mathit{SKIP} \;\mid$	Successful termination
	$\mathit{CHAOS} \;\mid$	Might do anything
	event $\rightarrow$ Process $\mid$	Event prefix
	channel?x $\rightarrow$ Process $\mid$	Input
	channel!x $\rightarrow$ Process $\mid$	Output
	Process$_1$; Process$_2$ $\mid$	Sequential composition
	Process$_1$ $\vert[\, \text{alph}_1 \Vert \text{alph}_2 \,]\vert$ Process$_2$ $\mid$	Parallel Composition
	Process$_1$ $\sqcap$ Process$_2$ $\mid$	Non-deterministic choice
	Process$_1$ $\Box$ Process$_2$ $\mid$	Deterministic choice
	if B **then** Process$_1$ **else** Process$_2$ $\mid$	Conditional
	Process $\setminus$ event $\mid$	Event hiding
	f(Process) $\mid$	Event relabelling
	name	

The meaning of a CSP process is defined in terms of the circumstances under which it might exhibit the phenomena of *deadlock* or *divergence*. This is the *Failures-Divergences* model. A process which is deadlocked is permanently blocked in a state where it is able neither to perform any event nor to terminate successfully. A program which is divergent is locked into an infinite pattern of concealed activity. To the outside world both phenomena appear the same.

The terminology for the failures-divergences model is defined as follows. A *trace tr* of a process P is any finite sequence of events $\langle e_1, e_2 \ldots e_n \rangle$ that it may perform from its initial state. A *divergence* of a process is a trace after which it *might* diverge. A *failure* of a process P consists of a pair (tr, ref) where tr is a trace of P and ref is a set of events which if offered to P by its environment after it has performed trace tr, might be completely refused.

Each CSP process is then uniquely defined by a pair of sets (F, D), corresponding to its *failures* and *divergences*.[1]

Precise definitions for the *failures* and *divergences* of CSP processes are given in [8] by equations such as

$$
\begin{aligned}
\mathit{divergences}(\mathit{STOP}) &= \emptyset \\
\mathit{failures}(\mathit{STOP}) &= \{\langle\rangle\} \times \mathbb{P}\,\Sigma \\
\mathit{divergences}(x \rightarrow P) &= \left\{\langle x\rangle \frown tr \mid tr \in \mathit{divergences}(P)\right\} \\
\mathit{failures}(x \rightarrow P) &= \{(\langle\rangle, X) \mid X \subseteq \Sigma - \{x\}\} \\
&\cup \left\{(\langle x\rangle \frown tr, X) \mid (tr, X) \in \mathit{failures}(P)\right\}
\end{aligned}
$$

An important characteristic of the model to be noted is that the possibility of divergence is always treated as being catastrophic. It is identified with the primitive process *CHAOS*

---

[1] The *traces* of a process may be fully determined from its *failures*.

which is the most completely unpredictable CSP process of all. This means that it is virtually impossible to prove anything useful about a divergent process. The main purpose for allowing for this form of behaviour in the model is so as to be able to prove its absence.

Let us introduce a couple of simple CSP processes here to illustrate the concepts covered in this section. First consider a process to represent a typical vending machine.

$$VM = coin \rightarrow tea \rightarrow VM \sqcap coin \rightarrow VM$$

This vending machine is faulty. It is supposed to accept a coin, then dispense a cup of tea. However sometimes it swallows up the coin without dispensing anything, quite unpredictably. Here are some possible traces for *VM*.

$$\begin{array}{r}\langle coin, tea, coin, tea, coin, tea\rangle \\ \langle coin, coin, coin, coin\rangle \\ \langle coin, tea, coin, coin\rangle \\ \langle coin, coin, tea, coin\rangle\end{array}$$

After the machine has performed trace $\langle coin\rangle$ it may or may not refuse to serve a cup of tea. This is recorded by the following two pieces of information.

$$\begin{array}{rcll}\langle coin, tea\rangle & \in & traces(VM) & \textit{...might}\text{ serve tea after receiving a coin} \\ (\langle coin\rangle, \{tea\}) & \in & failures(VM) & \textit{...might refuse}\text{ to serve tea after receiving a coin}\end{array}$$

Now let us consider a process to describe the behaviour of a consumer of hot drinks.

$$TD = coin \rightarrow tea \rightarrow TD \,\square\, coffee \rightarrow TD$$

The tea drinker is happy either to pay for cups of tea, or to drink coffee free of charge. He allows his environment to control this choice if necessary.

We can use the CSP parallel operator to combine the two processes *TD* and *VM* into a single process. To do this we need to specify for each process an *alphabet* to define the set of communication events in which it is required to participate. In this example it makes sense for the alphabet of *TD* to be $\{coin, coffee, tea\}$ and for the alphabet of *VM* to be $\{coin, tea\}$. We then write the parallel composition as

$$TD \;\; |[\, \{coin, coffee, tea\} \,||\, \{coin, tea\} \,]| \;\; VM$$

Writing Specifications for CSP Systems

The failures-divergences model is used for formal reasoning about the behaviour of concurrent systems defined by CSP equations. Hoare invented a simple notation for this purpose. We write

$$P\,\mathbf{sat}\,f(tr, ref)$$

to specify that all failures (tr, ref) of process P must satisfy the predicate f.

For instance the owner of vending machine *VM* might wish to specify that it must never dispense more cups of tea than have been paid for by the clause[2]

$$VM\,\mathbf{sat}\,tr \downarrow tea \leqslant tr \downarrow coin \tag{1}$$

This specification is satisfied by the current faulty definition of *VM*, but is clearly not satisfactory for the customer as he might not receive any tea for his money.

A more reasonable specification from the customer's point of view would be that the machine should alternate between accepting a coin and dispensing a cup of tea, i.e.

$$VM\,\mathbf{sat}\,0 \leqslant tr \downarrow coin - tr \downarrow tea \leqslant 1 \tag{2}$$

However this specification still does not guarantee that he will receive any tea for his money, as it does not rule out that the machine will deadlock immediately after receiving the coin.

Better is to specify that the vending machine cannot refuse to serve tea immediately after receiving a coin, by

$$VM\,\mathbf{sat}\ (head(reverse(tr)) = coin) \Longrightarrow tea \notin ref \tag{3}$$

However, this specification says nothing about the situation where a machine allows the customer to insert two or more coins, before serving him any tea. Should the additional coins be regarded as forfeit on account of stupidity, or should the customer be allowed to claim as many cups of tea as the number of inserted coins?

Note that specifications (2) and (3) do not hold for the current faulty definition of *VM*.

Another important property that involves specification on refusal sets is that of deadlock-freedom. Let Σ be the universe of communication events, then process *P* is deadlock-free if, and only if,

$$P\,\mathbf{sat}\,ref \neq \Sigma$$

Generally the only property we are likely to want to show about divergence is its absence, i.e. $divergences(P) = \{\}$, so this component of the formal model is not usually included as part of the **sat** language.

Proving **sat** *Clauses Algebraically*

Clearly the usefulness of **sat** specification clauses depends on the feasibility of proving their validity. One approach is to calculate directly the *failures* and *divergences* of the process definition to be analysed and then perform proofs regarding the contents of these mathematical objects. As these are likely to be highly complex and infinite sets, this method is unappealing. A second approach is to use a system of deduction rules, such as the following examples from a set of rules due to Hoare.

$$\begin{array}{rll} \text{If}\,P\,\mathbf{sat}\,S\,\text{and}\,P\,\mathbf{sat}\,T & \text{then} & P\,\mathbf{sat}\,S \wedge T \\ \text{If}\,P\,\mathbf{sat}\,S(tr) & \text{then} & x \rightarrow P\ \mathbf{sat}\ (tr = \langle\rangle \vee (head(tr) = x \wedge S(tail(tr)))) \end{array}$$

However this approach also seems destined to require complex and intricate analyses to be carried out.

This paper is concerned with providing a simple algorithm for *automated* verification of **sat** clauses, which will be introduced in section 4.

---

[2] The notation $tr \downarrow x$ means the number of times that event *x* occurs in trace *tr*. See the appendix for a glossary of trace functions.

The Refinement Approach

An alternative to the **sat** notation for specifying CSP processes is given by process refinement. Here required behaviour constraints may be specified as abstract, non-deterministic CSP processes.

There is a natural ordering on the set of all processes given by

$$(F_1, D_1) \sqsubseteq (F_2, D_2) \iff F_1 \supseteq F_2 \wedge D_1 \supseteq D_2$$

The interpretation of this is that process P_1 is worse than P_2 if it can deadlock or diverge whenever P_2 can. The worst process of all is *CHAOS*.

This ordering is very important to the stepwise refinement of concurrent systems. Starting from an abstract, non-deterministic definition, details of components may be independently fleshed out whilst preserving important properties of the overall system such as freedom from deadlock and divergence.

The refinement ordering provides an alternative approach to specifying the behaviour of CSP systems. To determine whether a process *Imp* satisfies a particular property *p* we construct the *worst possible* process *Spec* that satisfies *p* and then check that the process *Imp* refines *Spec* (or *Spec* is worse than *Imp*: *Spec* $\sqsubseteq$ *Imp*).

For example, in order to check that a process is divergence-free, we compare it with the worst possible divergence-free process, *DIVFREE*, given by

$$DIVFREE = STOP \sqcap \left(\Box_{x:\Sigma} x \rightarrow DIVFREE \right)$$

This process may perform any event (from the global universe Σ) at any time, or it may deadlock at any time. But it will never diverge.

Using refinement, specification statement (2) from the previous section becomes

$$VM \sqsupseteq DIVFREE \;|[\, \Sigma \,||\, \{tea, coin\} \,]|\; ALTERNATE^3$$

Where

$$ALTERNATE = coin \rightarrow tea \rightarrow ALTERNATE$$

The parallel composition of *DIVFREE* and *ALTERNATE* defines a process which never diverges but may deadlock at any time, and the only constraint on the events that it may perform is that alternation is required between *coin* and *tea* starting with *coin*[4]. (The reader will find that it is rather more difficult to express clauses (1) and (3) as refinement assertions.)

A significant advantage of using refinement expressions over **sat** clauses until now has been the existence of an automated tool for checking their validity – the FDR tool of Formal Systems Europe[2]. The intention of this paper is to prepare the ground for the development of a similar tool for the verification of **sat** clauses, which appear to provide a more expressive notation.

3 Normal Form Transition Systems

It will be observed that the failures sets for most interesting processes will be infinite, certainly for any non-terminating process. For automated analysis a compact finite representation is required. This is given by the *normal form transition system* devised by A.W.Roscoe for use in the refinement checking program FDR.

---

[3] This assertion is, of course, false for the current definition of *VM*.

[4] It was necessary to include divergence-freedom in the specification process because CSP identifies divergence with *CHAOS* – a process that does not satisfy any reasonable constraints.

Here any process P, with a finite number of recognisable states, is represented by a transition system $NF(P)$. Each state of $NF(P)$ corresponds to a set of traces of P after which the subsequent behaviour is identical. So we have effectively defined an equivalence relation $\sim$ on $traces(P)$ given by

$$tr_1 \sim tr_2 \iff P\,\mathrm{after}\,tr_1 = P\,\mathrm{after}\,tr_2$$

If there exist any divergent traces of P, *i.e.* $divergences(P) \neq \emptyset$, then they belong to a single equivalence class which corresponds to a state labelled with a flag $\perp$. States which correspond to non-divergent classes of traces are labelled with a set of *minimal acceptance sets* $\{A_1, .., A_m\}$. Minimal acceptance sets are the complement in Σ of maximal refusal sets[5]. They will clearly be the same for any particular class of equivalent traces. Minimal acceptances are used instead of refusals because they usually require less storage space.

The transitions of $NF(P)$ are defined as follows. There is a transition $\mathrm{state}_1 \xrightarrow{x} \mathrm{state}_2$ if and only if for every trace tr represented by state_1 there is a corresponding trace $tr \frown \langle x \rangle$ which is represented by state_2.

The normal form transition systems for processes VM and TD, defined in section 2, are illustrated in figure 1. Observe that the action of the nondeterministic choice operator $\sqcap$ is absorbed into state 1 of $NF(VM)$. The nondeterminism is represented by the presence of two distinct minimal acceptance sets.

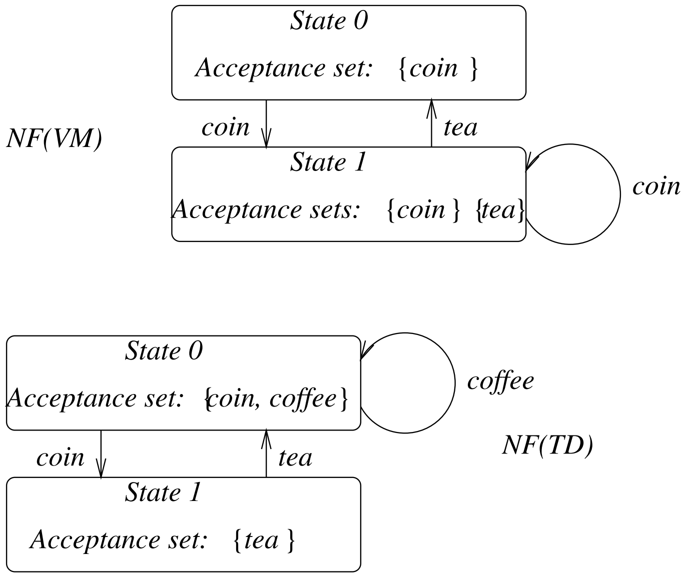

Figure 1: Normal Form Transition Systems

The algorithm by which FDR calculates normal-form transition systems is described in [7] and [4].

4 A Checking Algorithm for sat

Complete information about failures and divergences of a process may be extracted from its normal-form transition system. Generally if a process may diverge then we shall not be very interested in proving anything useful about it, so from now on we shall assume that all

---

[5] In the CSP failures-divergences model any subset of a refusal set is also a refusal set.

processes considered are divergence-free. (This property is checked during the construction of a normal-form transition system.) It will also be useful, from now on, to assume that the *ref* variable in a **sat** clause refers only to the *maximal* refusal sets corresponding to any given trace *tr*. This does not cause any loss of generality and yet makes the checking process far more efficient.

Specifications on refusal sets are easy to check because all the required information may be deduced from the list of minimal acceptance sets stored at each vertex. Each vertex needs to be looked at only once, since it represents an equivalence class of *all* traces after which the process behaves in a particular way. For instance, to check that a process is deadlock-free, i.e. $P\,\mathbf{sat}\,ref \neq \Sigma$, it would suffice to check that no state of $NF(P)$ is labelled with an empty acceptance set.

However checking a trace specification might potentially lead to an infinite search unless the specification is carefully stated.

Consider the specification

$$P\,\mathbf{sat}\,(tr \downarrow b + 4) \geqslant 2(tr \downarrow a) \geqslant tr \downarrow b$$

Starting at the initial state of P we might search through the transition digraph, keeping a record of the current trace, and checking every possible trace for $tr \downarrow a$ and $tr \downarrow b$. This search might never terminate for a non-terminating process, as the values of both $tr \downarrow a$ and $tr \downarrow b$ might increase continually, in step with each other.

There is a much better approach to this problem, as follows. We write our specification like this

$$P\,\mathbf{sat}\,4 \geqslant 2(tr \downarrow a) - tr \downarrow b \geqslant 0$$

Then we define an *incremental* trace function f as follows

$$f(\langle\rangle) = 0$$

$$f(tr ^\frown \langle x\rangle) \;=\; \begin{cases} f(tr) + 2 & \text{if} \quad x = a \\ f(tr) - 1 & \text{if} \quad x = b \\ f(tr) & \text{otherwise} \end{cases}$$

It is clear that

$$f(tr) = 2(tr \downarrow a) - tr \downarrow b$$

We start an exhaustive search through the transition system for pairs of the form (σ, v), where σ is a state and v is a possible value of $f(tr)$ at that state. The search terminates either when there are no new such pairs to be found, or if we find a pair for which $\neg\ (4 \geqslant v \geqslant 0)$.

There are two reasons why this approach is better. Firstly we have defined our variant function, f, in an incremental way, which means that we do not need to store any information about traces. The value of $f(tr)$ at each point in the search can be calculated purely from the information stored at the previous point. Secondly we have converted an endless search into one that is guaranteed to terminate, due to the bounds placed on the range of f.

This technique can be extended to a parallel network of two (or more) processes $V = P\ \|[\,\alpha P \,\|\, \alpha Q\,]|\ Q$, and a specification on network states $(tr, \langle ref_P, ref_Q\rangle)$, where ref_P and ref_Q are refusals of the individual processes P and Q after the network has performed trace *tr*. We now assume that the specification is expressed in the form

$$V\,\mathbf{sat}\,PRED(f_1(tr), \ldots, f_n(tr), ref_P, ref_Q)$$

involving a number of incremental trace functions f_i and refusal sets ref_P and ref_Q of P and Q.

Two sets of records are maintained: *pending* and *done*. Each record is of the form

$$(\sigma_P, \sigma_Q, v_1, .., v_n)$$

where (σ_P, σ_Q) is a pair of normal form states in which P and Q may simultaneously rest, and each v_i is the value of $f_i(tr)$ for a corresponding trace tr. The algorithm proceeds as follows.

1. Initially *pending* consists of a single record corresponding to the original state of the system, and *done* is empty.

   $$\begin{array}{rcl} pending & := & \{(0, 0, f_1(\langle\rangle), .., f_n(\langle\rangle))\} \\ done & := & \{\} \end{array}$$

   (We are assuming that the initial state of each process is numbered 0.)

2. Take a new record from *pending* to be processed.

   $$\begin{array}{rcl} r & := & (\sigma_P, \sigma_Q, v_1, .., v_n) \in pending \\ pending & := & pending - \{r\} \end{array}$$

3. Now check whether record r satisfies the specification. Suppose that σ_P has a set A of minimal acceptance sets and σ_Q has a set B of minimal acceptance sets.

   If $\exists\, a : A, b : B. \quad \neg PRED(v_1, .., v_n, \alpha P - a, \alpha Q - b)$ then halt. (The specification is *not* satisfied). Otherwise

   $$done := done \cup \{r\}$$

4. Now construct the set *new* of successor records of r, by considering every transition that is possible for $P \;|[\,\alpha P \,||\, \alpha Q\,]|\; Q$ from state pair (σ_P, σ_Q). Assume that r corresponds to some trace tr of $P \;|[\,\alpha P \,||\, \alpha Q\,]|\; Q$. Then

   $$\begin{array}{rcl} new & := & \left\{ \begin{array}{c} (\sigma'_P, \sigma_Q, f_1(tr ^\frown \langle x\rangle), .., f_n(tr ^\frown \langle x\rangle)) \mid \\ x \in \alpha P - \alpha Q \wedge \sigma_P \xrightarrow{x} \sigma'_P \end{array} \right\} \\ & \cup & \left\{ \begin{array}{c} (\sigma_P, \sigma'_Q, f_1(tr ^\frown \langle x\rangle), .., f_n(tr ^\frown \langle x\rangle)) \mid \\ x \in \alpha Q - \alpha P \wedge \sigma_Q \xrightarrow{x} \sigma'_Q \end{array} \right\} \\ & \cup & \left\{ \begin{array}{c} (\sigma'_P, \sigma'_Q f_1(tr ^\frown \langle x\rangle), .., f_n(tr ^\frown \langle x\rangle)) \mid \\ x \in \alpha P \cap \alpha Q \wedge \sigma_P \xrightarrow{x} \sigma'_P \wedge \sigma_Q \xrightarrow{x} \sigma'_Q \end{array} \right\} \end{array}$$

   Although we have not stored any record of a value of tr that corresponds to r, it is not actually required in order to perform this calculation due to the incremental method of defining the various trace functions.

5. Now we eliminate records from *new* that have already been processed and merge the remainder into *pending*.

   $$pending := pending \cup (new - done)$$

6. If $pending = \{\}$ then halt. (The specification is satisfied.) Otherwise return to step 2.

This algorithm is not certain to terminate for every given set of incremental trace functions f_i and predicate *PRED*. But if there is a finite range of values for each f_i outside which satisfaction of *PRED* is impossible then termination is guaranteed for any network.

The following example is included in order to illustrate this technique. Consider the network $V = \mathit{LEFT}\ \|[\ \{\mathit{in,mid}\} \,\|\, \{\mathit{mid,out}\}\ ]\|\ \mathit{RIGHT}$ with the following process definitions.

$$\begin{array}{rcl} \mathit{LEFT} & = & \mathit{in} \rightarrow \mathit{mid} \rightarrow \mathit{LEFT} \\ \mathit{RIGHT} & = & \mathit{mid} \rightarrow \mathit{out} \rightarrow \mathit{RIGHT} \end{array}$$

Suppose we wish to prove that the following trace specification is satisfied.

$$V\ \mathbf{sat}\ 2 \geqslant tr \downarrow in - tr \downarrow out \geqslant 0$$

V is an abstract representation of a double buffer, which inputs information on channel *in* and outputs it on channel *out*. The specification simply states that the number of messages held in the buffer at any given time lies between nought and two inclusive.

We proceed by defining an incremental trace function f as follows

$$\begin{array}{rcl} f(\langle\rangle) & = & 0 \\ f(tr \frown \langle x \rangle) & = & \begin{cases} f(tr)+1 & \text{if} \quad x = in \\ f(tr)-1 & \text{if} \quad x = out \\ f(tr) & \text{otherwise} \end{cases} \end{array}$$

It is clear that

$$f(tr) = tr \downarrow in - tr \downarrow out$$

In this case our predicate function *PRED* is given by

$$\mathit{PRED}(f(tr)) = (2 \geqslant f(tr) \geqslant 0)$$

Normal form state transition systems for the network V are shown in figure 2. We now proceed to form an exhaustive set of records of the form

$$(\sigma_{LEFT}, \sigma_{RIGHT}, val)$$

consisting of a state of process *LEFT*, a corresponding state of process *RIGHT* and a possible value for $f(tr)$ when the processes are in those states.

The search proceeds as follows. First we have

$$pending = \{(0,0,0)\}, \quad done = \{\}$$

Check $(0,0,0)$; possible transition is *in*; leads to record: $(1,0,1)$. Now we have

$$pending = \{(1,0,1)\}, \quad done = \{(0,0,0)\}$$

Check $(1,0,1)$; possible transition is *mid*; leads to record: $(0,1,1)$. Now we have

$$pending = \{(0,1,1)\}, \quad done = \{(0,0,0),(1,0,1)\}$$

Check $(0,1,1)$; possible transitions are *in, out*; lead to records: $(1,1,2)$, $(0,0,0)$. Now we have

$$pending = \{(1,1,2)\}, \quad done = \{(0,0,0),(1,0,1),(0,1,1)\}$$

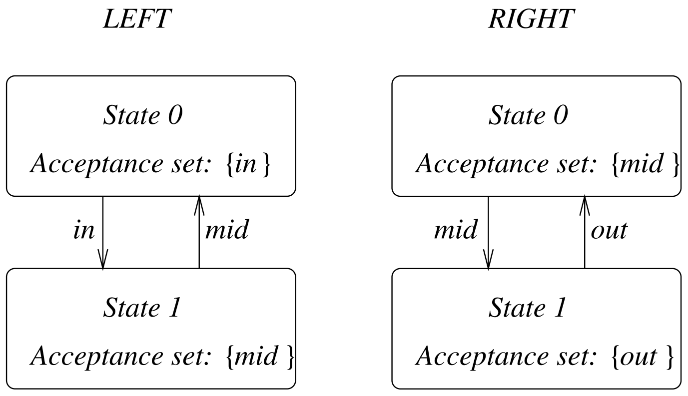

Figure 2: Normal Form Transition Systems for Two-Place Buffer

Check $(1,1,2)$; possible transition is *out*; leads to record: $(1,0,1)$. Now we have

$$pending = \{\}, \quad done = \{(0,0,0),(1,0,1),(0,1,1),(1,1,2)\}$$

The search is now complete. Every record that was found satisfies the original specification, and we shall conclude that it is satisfied by V. This is rather a bold claim given that the set of traces of V is infinite and we have only examined four cases. But it may be justified by using induction on traces, as follows.

Every trace *tr* of V corresponds to a unique pair of normal-form states

$$(\sigma_{LEFT}, \sigma_{RIGHT})$$

These are found by constructing the unique walk in the normal-form transition system of *LEFT* with labels $tr \upharpoonright \alpha LEFT$, and the unique walk in the normal-form transition system of *RIGHT* with labels $tr \upharpoonright \alpha RIGHT$. We shall call this state pair

$$(\sigma_{LEFT}(tr), \sigma_{RIGHT}(tr))$$

Now suppose that for a certain trace *t*, we know that record

$$(\sigma_{LEFT}(t), \sigma_{RIGHT}(t), f(t))$$

lies in set *done*, constructed above. Now consider a trace $t \frown \langle x \rangle$ of V. This corresponds to a state pair

$$(\sigma_{LEFT}(t \frown \langle x \rangle), \sigma_{RIGHT}(t \frown \langle x \rangle))$$

which must be reachable from $(\sigma_{LEFT}, \sigma_{RIGHT})$ by one or both of the processes performing event x.

We have already assumed that $(\sigma_{LEFT}(t), \sigma_{RIGHT}(t), f(t))$ lies in set *done*. Therefore it must have at some point been selected from set *pending* at step 2 of the checking algorithm and record

$$(\sigma_{LEFT}(t \frown \langle x \rangle), \sigma_{RIGHT}(t \frown \langle x \rangle), f(t \frown \langle x \rangle))$$

must have been discovered at step 4 and so now also must lie in set *done*, given that the algorithm has terminated.

We actually know that

$$(\sigma_{LEFT}(\langle\rangle), \sigma_{RIGHT}(\langle\rangle), f(\langle\rangle)) = (0,0,0) \in done$$

because this is the record that was used to start the search. Hence, by induction, *every* trace *tr* of V is represented in *done* by a record of the form

$$(\sigma_{LEFT}(tr), \sigma_{RIGHT}(tr), f(tr))$$

So we conclude that the original specification is satisfied by all traces of V.

Although this proof technique is tedious for humans it is very easy to automate on a computer.

5 Some Examples of Incremental Trace Functions

Incremental trace functions are found to be surprisingly useful in the information that they can be made to carry. There now follow some simple examples.

Length of trace modulo n ($\sharp tr$ modulo n)

$$\begin{aligned} f(\langle\rangle) &= 0 \\ f(tr \frown \langle x\rangle) &= f(tr) + 1 \,\mathbf{modulo}\, n \end{aligned}$$

Trailing subsequence of tr of length n

$$\begin{aligned} f(\langle\rangle) &= \langle\rangle \\ f(tr \frown \langle x\rangle) &= \begin{cases} f(tr) \frown \langle x\rangle & \text{if} \quad \sharp f(tr) < n \\ tail(f(tr)) \frown \langle x\rangle & \text{otherwise} \end{cases} \end{aligned}$$

Number of events following last occurrence of event e

$$\begin{aligned} f(\langle\rangle) &= null \\ f(tr \frown \langle x\rangle) &= \begin{cases} 0 \quad \text{if} \quad x = e \\ f(tr) + 1 \quad \text{if} \quad x \neq e \,\text{and}\, f(s) \neq null \\ null \quad \text{otherwise} \end{cases} \end{aligned}$$

i*th event of *tr

$$\begin{aligned} f(\langle\rangle) &= (0, null) \\ f(tr \frown \langle x\rangle) &= \begin{cases} (i+1, x) \quad \text{if} \quad left(f(tr)) = i \\ (left(f(tr)) + 1, null) \quad \text{if} \quad left(f(tr)) < i \\ f(tr) \quad \text{otherwise} \end{cases} \end{aligned}$$

Here *left* and *right* are standard tuple projection functions and the value of $tr[i]$ is given by $right(f(tr))$.

6 Case Studies

A Simple Railway Signalling System

The verification technique described in this paper could potentially be used for guaranteeing safety in a railway system. Figure 3 shows a simple single track railway circuit, with two driverless trains, such as might be found at an amusement park. The track is divided into three distinct segments guarded by signals, and, in order to avoid collisions, it is important to ensure that the two trains can never be on the same segment of track simultaneously.

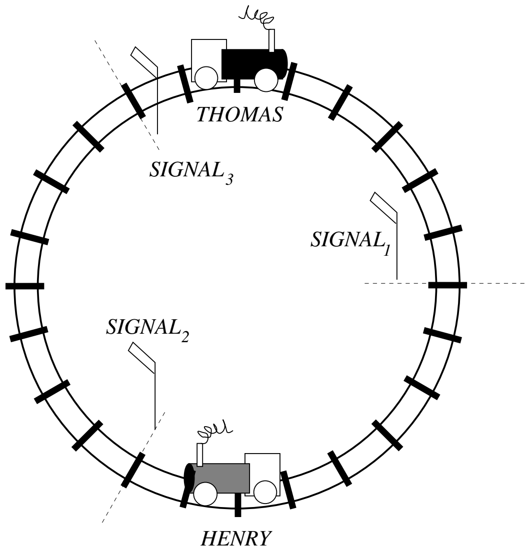

Figure 3: Simple Railway Signalling System

We shall model the system as a network V of five processes:

$$\langle THOMAS, HENRY, SIGNAL_1, SIGNAL_2, SIGNAL_3 \rangle$$

We use names $enter.T.i$ and $enter.H.i$, where i ranges between 1 and 3, to represent the events of 'Thomas' and 'Henry' entering particular segments of the track.

If we assume that, when the system first comes into operation, Thomas is situated in the segment of track guarded by $SIGNAL_3$ and Henry is situated in the segment guarded by $SIGNAL_1$ we can model the safety condition as follows:

$$\begin{array}{rcl}
V & \mathbf{sat} & ThomasSegment(tr) \neq HenrySegment(tr) \\
\text{where}\quad ThomasSegment(\langle\rangle) & = & 3 \\
ThomasSegment(tr \frown \langle x \rangle) & = & \begin{cases} 1\,\text{if}\, x = enter.T.1 \\ 2\,\text{if}\, x = enter.T.2 \\ 3\,\text{if}\, x = enter.T.3 \\ ThomasSegment(tr)\text{otherwise} \end{cases} \\
\text{and}\quad HenrySegment(\langle\rangle) & = & 1 \\
HenrySegment(tr \frown \langle x \rangle) & = & \begin{cases} 1\,\text{if}\, x = enter.H.1 \\ 2\,\text{if}\, x = enter.H.2 \\ 3\,\text{if}\, x = enter.H.3 \\ HenrySegment(tr)\text{otherwise} \end{cases}
\end{array}$$

A possible implementation of the network is as follows:

$$\begin{aligned}
\mathit{THOMAS} &= \mathit{enter}.T.1 \rightarrow \mathit{enter}.T.2 \rightarrow \mathit{enter}.T.3 \rightarrow \mathit{THOMAS} \\
\alpha\mathit{THOMAS} &= \{\mathit{enter}.T.1, \mathit{enter}.T.2, \mathit{enter}.T.3\} \\
\mathit{HENRY} &= \mathit{enter}.H.2 \rightarrow \mathit{enter}.H.3 \rightarrow \mathit{enter}.H.1 \rightarrow \mathit{HENRY} \\
\alpha\mathit{HENRY} &= \{\mathit{enter}.H.1, \mathit{enter}.H.2, \mathit{enter}.H.3\} \\
\mathit{SIGNAL}_1 &= \mathit{ready}.2 \rightarrow \mathit{enter}.T.1 \rightarrow \mathit{ready}.1 \rightarrow \mathit{ready}.2 \rightarrow \\
&\quad\ \mathit{enter}.H.1 \rightarrow \mathit{ready}.1 \rightarrow \mathit{SIGNAL}_1 \\
\alpha\mathit{SIGNAL}_1 &= \{\mathit{enter}.T.1, \mathit{enter}.H.1, \mathit{ready}.2, \mathit{ready}.1\} \\
\mathit{SIGNAL}_2 &= \mathit{enter}.H.2 \rightarrow \mathit{ready}.2 \rightarrow \mathit{ready}.3 \rightarrow \mathit{enter}.T.2 \rightarrow \\
&\quad\ \mathit{ready}.2 \rightarrow \mathit{ready}.3 \rightarrow \mathit{SIGNAL}_2 \\
\alpha\mathit{SIGNAL}_2 &= \{\mathit{enter}.T.2, \mathit{enter}.H.2, \mathit{ready}.3, \mathit{ready}.2\} \\
\mathit{SIGNAL}_3 &= \mathit{ready}.1 \rightarrow \mathit{enter}.H.3 \rightarrow \mathit{ready}.3 \rightarrow \mathit{ready}.1 \rightarrow \\
&\quad\ \mathit{enter}.T.3 \rightarrow \mathit{ready}.3 \rightarrow \mathit{SIGNAL}_3 \\
\alpha\mathit{SIGNAL}_3 &= \{\mathit{enter}.T.3, \mathit{enter}.H.3, \mathit{ready}.1, \mathit{ready}.3\}
\end{aligned}$$

Here we are using the channels *ready*.1, *ready*.2 and *ready*.3 as the means of communication between the signalling processes. The safety property may easily be verified for our implementation of the network using the incremental function technique described above[6].

This example has a close relationship with proving correctness of networking protocols using parallel programming languages that are based on CSP, such as occam.

A Proof Rule of Brookes and Roscoe

Another application of the incremental function technique is given by automation of a proof technique for deadlock-freedom due to S.D.Brookes and A.W.Roscoe[1], which has been used in the design of routing protocols.

We consider a network of processes $V = \langle P_1 \,..\, P_n \rangle$ composed in parallel. Each process P_i has alphabet αP_i. We shall need to recall a little deadlock-analysis terminology.

Triple-Disjoint A network is *triple-disjoint* if no event is shared by the alphabet of more than two processes.

Busy A network is *busy* if every component process P_i is individually deadlock-free.

Vocabulary The network *vocabulary* $\Lambda(V)$ is defined as the set of shared events $\bigcup_{i \neq j} \alpha P_i \cap \alpha P_j$.

Ungranted Request In network state $(tr, \langle ref_{P_1} \,..\, ref_{P_n} \rangle)$ P_i has an *ungranted request* to P_j whenever P_i wishes to communicate with P_j but P_j refuses to accept any communication that P_i offers, and neither process can communicate outside the network vocabulary.

Strong-Conflict-Free A network is *strong-conflict-free* if whenever there are two processes P_i and P_j, each one having an ungranted request to the other, then both processes are also able to communicate with some other process.

---

[6] The network may also be proven deadlock-free using the Deadlock Checker program[5].

The proof rule may be stated as follows:

> Let $V = \langle P_1 \mathinner{..} P_n \rangle$ be a busy, triple-disjoint, strong-conflict-free network such that whenever a process P has an ungranted request to another process Q then Q has previously communicated with P, and has done so more recently than with any other process. It follows that V is deadlock-free.

The properties of triple-disjointedness, and business are simple to check (see [4]). The remaining conditions of this proof rule may be expressed as follows.

$$\forall\, i,j.\quad P_i \;|[\, \alpha P_i \,\|\, \alpha P_j \,]|\; P_j \,\mathbf{sat} \begin{pmatrix} \neg \mathit{StrongConflict}_{ij}(\mathit{ref}_{P_i}, \mathit{ref}_{P_j}) \wedge \\ \mathit{UngrantedRequest}_{ij}(\mathit{ref}_{P_i}, \mathit{ref}_{P_j}) \Longrightarrow \\ (\mathit{LastComm}_j(tr) = i) \end{pmatrix}$$

Where

$$\begin{array}{rcl} \mathit{UngrantedRequest}_{ij}(\mathit{ref}_{P_i}, \mathit{ref}_{P_j}) & = & \begin{array}{c} ((\alpha P_i - \mathit{ref}_{P_i}) \cap \alpha P_j \neq \{\}) \wedge \\ (\alpha P_i \cap \alpha P_j \subseteq \mathit{ref}_{P_i} \bigcup \mathit{ref}_{P_j}) \wedge \\ ((\alpha P_i - \mathit{ref}_{P_i}) \cup (\alpha P_j - \mathit{ref}_{P_j}) \subseteq \Lambda(V)) \end{array} \\ \mathit{StrongConflict}_{ij}(\mathit{ref}_{P_i}, \mathit{ref}_{P_j}) & = & \begin{array}{c} \mathit{UngrantedRequest}_{ij}(\mathit{ref}_{P_i}, \mathit{ref}_{P_j}) \wedge \\ \mathit{UngrantedRequest}_{ji}(\mathit{ref}_{P_j}, \mathit{ref}_{P_i}) \wedge \\ ((\alpha P_i - \mathit{ref}_{P_i} \subseteq \alpha P_j) \vee (\alpha P_j - \mathit{ref}_{P_j} \subseteq \alpha P_i)) \end{array} \\ \mathit{LastComm}_j(\langle\rangle) & = & \mathit{null} \\ \mathit{LastComm}_j(tr \frown \langle x \rangle) & = & \begin{cases} \mathit{LastComm}_j(tr) & \text{if} \quad x \notin \Lambda(V) \cap \alpha P_j \\ i \quad \text{such that} \quad x \in \alpha P_i \wedge i \neq j & \text{otherwise} \end{cases} \end{array}$$

It will be seen that this expression is of a suitable form for the application of the checking algorithm described in this paper. (In fact a simplified version of this check is implemented as part of the Deadlock-Checker program[5].)

7 Conclusions and Future Prospects

This paper has described a technique for verifying CSP processes, and parallel networks of processes, against specifications expressed as **sat** clauses. This approach seems to be more expressive and powerful than refinement. However the new technique is not fully general. It applies only to particular types of specifications that may be expressed using bounded, incremental functions. Further work is required to explore fully the power and limitations of this technique

The next step would be to develop a tool for automatic verification of **sat** clauses. The most difficult aspect of this would be the process of deriving from a given clause a suitable predicate involving incremental functions. A library of useful incremental functions, such as those listed in the previous sections, would need to be provided to act as building blocks for this construction. It seems likely that one would have to enforce restrictions on the syntax of those **sat** clauses that could be checked.

We illustrated how the incremental function technique can be extended to analyse networks of processes, in order to verify specifications that involve the refusal sets of individual processes within the network. The incremental function approach to verifying CSP processes first emerged in work to check deadlock-freedom[4] through adherence to design rules (e.g. see [6]). These design rules sometimes incorporate protocol specifications which are specified in terms of the refusal sets of individual processes, and such specifications cannot be checked directly using the refinement approach of FDR, but can be checked using the technique described in this paper.

Applying our checking technique to large networks of processes is difficult because of the problem of exponential state explosion – the number of states of a network usually grows exponentially with the number of parallel components. To combat this there is scope for developing a hierarchical proof system for the **sat** language. The idea would be that a specification clause of a large network might be derived from a collection of specifications involving small subnetworks which would be feasible to check. For instance, we might check a clause $P \;|[\,\alpha P \,||\, \alpha Q\,]|\; Q \,\mathbf{sat}\, f(tr, ref)$ by proving

$$P\,\mathbf{sat}\,g(tr, ref)\,\mathbf{and}\,Q\,\mathbf{sat}\,h(tr, ref) \quad \Longrightarrow \quad P \;|[\,\alpha P \,||\, \alpha Q\,]|\; Q\,\mathbf{sat}\,f(tr, ref)$$

and then checking $P\,\mathbf{sat}\,g(tr, ref)$ and $Q\,\mathbf{sat}\,h(tr, ref)$. We could attempt to guarantee the safety property of a complex railway system by a collection of local analyses of small regions of track. Such an approach would be a natural extension to the automatic technique for proving deadlock-freedom described here and in [4].

References

[1] S. D. Brookes and A. W. Roscoe *Deadlock Analysis in Networks of Communicating Processes*, Distributed Computing (1991)4, Springer Verlag

[2] *FDR User Manual and Tutorial* Formal Systems (Europe) Ltd. 3 Alfred Street, Oxford OX1 4EH. Available at `ftp://ftp.comlab.ox.ac.uk/pub/Packages/FDR`

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

[4] J. M. R. Martin *The Design and Construction of Deadlock-Free Concurrent Systems*. D. Phil. Thesis, University of Buckingham 1996.
(also available at `http://www.hensa.ac.uk/parallel/theory/formal/csp`)

[5] J. M. R. Martin and S. A. Jassim, *A Tool for Proving Deadlock Freedom*, Proceedings of 20th World Occam and Transputer User Group Technical Meeting (1997), IOS Press

[6] J. M. R. Martin and P. H. Welch *A Design Strategy for Deadlock-Free Concurrent Systems* Transputer Communications Volume 3 Number 4 (1996)

[7] A. W. Roscoe *Model Checking CSP*, A Classical Mind, Prentice Hall 1994.

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

Appendix A: Glossary of Trace Terminology

$\langle\rangle$ The empty trace

$\langle a \rangle$ The singleton sequence containing only a

$tr_1 \frown tr_2$ Trace tr_1 concatenated with trace tr_2

tr^n Trace tr repeated n times

$tr \upharpoonright A$ Trace tr restricted to events in set A

$tr_1 \ \mathbf{in} \ tr_2$ Trace tr_1 lies within trace tr_2

$tr_1 \leqslant tr_2$ Trace tr_1 is a prefix of trace tr_2

$\sharp tr$ Length of trace tr

$tr \downarrow a$ Number of occurrences of a in tr

$head(tr)$ The first element of tr

$tail(tr)$ The result of removing the first element from tr

$reverse(tr)$ The result of reversing the order of the elements of tr

Appendix B: Relationship between occam and CSP

We anticipate that some readers of this paper will be rather more familiar with occam than with CSP. Table 1 lists some roughly equivalent constructions between the languages and will hopefully clarify much of the above material for non-CSP-specialists.

The CSP processes *VM* and *TD*, and their parallel composition, might be 'implemented' in occam as shown in figure 4. However it should be noted that most implementations of occam do not allow output guards within `ALT` constructs. This is done for reasons of efficiency.

```
PROC VM (CHAN OF SIGNAL coin, tea)
  WHILE TRUE
    ALT
      SIGNAL any:
      TRUE & SKIP
        SEQ
          coin ? any
          tea ! any
      SIGNAL any:
      TRUE & SKIP
        coin ? any
:

PROC TD (CHAN OF SIGNAL coin, tea, coffee)
  WHILE TRUE
    ALT
      SIGNAL any:
      coin ! any
        tea ? any
      SIGNAL any:
      coffee ? any
        SKIP
:

CHAN OF SIGNAL coin, tea, coffee:
PAR
  TD (coin, tea, coffee)
  VM (coin, tea)
```

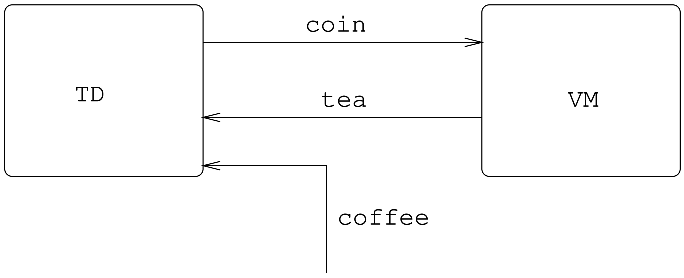

Figure 4: occam code for *VM* and *TD* processes

occam	**CSP**

`SEQ`
`  P`
`  Q`
	$P;\ Q$

`PAR`
`  P`
`  Q`
	$P \ \vert[\, \alpha P \,\Vert\, \alpha Q \,]\vert\ Q$

`a?x`	$a?x \rightarrow SKIP$
`b!y`	$b!y \rightarrow SKIP$

`ALT`
`  c?x`
`    P`
`  d?y`
`    Q`
	$c?x \rightarrow P \ \Box$
	$d?y \rightarrow Q$

`ALT`
`  TRUE & SKIP`
`    P`
`  TRUE & SKIP`
`    Q`
	$P \sqcap Q$

`IF`
`  b`
`    P`
`  NOT b`
`    Q`
	if b **then** P **else** Q

`WHILE TRUE` `  P`	Process X such that $X = P;\ X$

T