Available online at www.sciencedirect.com
ScienceDirect
Electronic Notes in Theoretical Computer Science 250 (2009) 171–186
Electronic Notes in Theoretical Computer Science
www.elsevier.com/locate/entcs

Counter Abstraction in the CSP/FDR setting

Tomasz Mazur[1] Gavin Lowe[2]

Oxford University Computing Laboratory
Wolfson Building, Parks Road, Oxford OX1 3QD, UK

Abstract

In this paper we consider an adaptation of counter abstraction for the CSP/FDR setting. The technique allows us to transform a concurrent system with an unbounded number of agents into a finite-state abstraction. The systems to which the method can be applied are composed of many identical node processes that run in parallel with a controller process. Refinement checks on the abstract state machine can be performed automatically in the traces and stable failures models using the FDR model checker. We illustrate the method on an example based on a multiprocessor operating system.

Keywords: Counter abstraction, model checking, parameterised verification, CSP.

1 Introduction

One subclass of a *Parameterised Model Checking Problem* can be specified as follows:

> *Given a concurrent system System(N), consisting of N identical and independent processes, and a specification Spec, is it true that System(N) satisfies Spec for all N?*

It has been shown by Apt and Kozen [1] that this problem is in general undecidable. The best we can do is to restrict the verification to a particular class of systems and provide a method that is sound (but not necessarily complete) for this class. The systems that we are going to consider consist of a number of identical *node processes*. Given N, the number of node processes present in the system, the general form of such systems is

$$System(N) = F(Nodes(N)),$$

where $F(\ )$ is any CSP context,

$$Nodes(N) = |||_{i=0}^{N-1} Node,$$

[1] Email: tomasz.mazur@comlab.ox.ac.uk
[2] Email: gavin.lowe@comlab.ox.ac.uk

doi:10.1016/j.entcs.2009.08.012

and $Node$ models a single node process.

We do not allow node process identifiers anywhere in the definition of $Node$: otherwise, the size of the alphabet of the implementation process would depend on N, and, because counter abstraction does not change alphabets, the abstraction of the system would also depend on N, contrary to our requirements. In addition, for the same reasons, we assume that the specification process $Spec$ and the CSP context $F(\_)$ are independent of N.

In most practical situations we can expect the context F to describe some controller process that is put in parallel with the nodes. The uniform verification of the family of problems with such a context has been proven by German and Sistla to be decidable [7]. However, the algorithm provided is double exponential and therefore the verification is only possible for very small instances of a problem.

Counter abstraction is already a well-known abstraction method, probably best presented by Pnueli et al. in [14]. It is an application of the more general *predicate abstraction* [4,6,12], which in turn is a special case of *finitary abstraction* [10], a method which transforms an infinite-state parameterised concurrent system into a finite-state abstraction. Counter abstraction aims to create an abstraction of the system that is independent of the number of node processes, to ensure that a single refinement check will prove a given result for all values of N. The main idea is to replace the concrete state space representing a concurrent system consisting of many similar processes by an abstract state space; each abstract state is a tuple of integer counters $(c_0, c_1, \ldots, c_{k-1})$, where each c_j counts how many node processes are currently in the j-th state. The counters are also given a threshold z and we interpret $c_j = z$ to mean that there are z *or more* processes in state j.

In the past, counter abstraction has been successfully applied to systems where the verification property has been given in the form of a temporal formula [2,14,18,19]. In [2], Clarke and Grümberg show that if $System(k) \parallel Abstr(System)$ and $System(k+1) \parallel Abstr(System)$ are equivalent under an appropriate relation, then it is enough to consider only the instances of the systems of size less than or equal to k in order to verify whether a given ICTL* formula holds *for all* sizes of the system. A method similar to counter abstraction with a threshold $z = 1$ is used to construct $Abstr(System)$. However, in order for the method to work, the user needs to provide two functions that match three special conditions. This task usually requires some ingenuity, so the scope for automation is limited.

Pnueli et al. in [14] and Xu in [19] show how to verify a general temporal formula in a concurrent system with unboundedly many similar processes running in parallel. The model used is Fair Transition Systems [13] where the processes are interleaved and communicate with each other via shared variables only. Each process may also be given its own local variables (whose values may depend on the process's identity). However, this destroys the full symmetry of the system. Before the verification procedure is performed, the user has to replace the local variables by some appropriate global variables that will model the same behaviour. Again, this task may require quite a lot of ingenuity. In the paper it is shown how, given a concrete state space and a temporal formula, to create a corresponding abstract machine and abstract temporal formula. Also some heuristics are provided for extending the abstract formula so that liveness can be proven.

The work that is probably the closest to ours is [18]. It considers systems of a similar structure to the one we use, and uses a model with synchronous communication between processes. The verification algorithm consists of two parts. The first creates a finite abstraction, which is independent of the number of processes present in the concrete system. This is similar to our method, but instead of using a fixed threshold z (introduced in Section 4), it determines appropriate thresholds on-the-fly (at the expense of compilation complexity). The second part of the algorithm evaluates the CTL-X specification on the abstract state machine.

The novelty of our work is the application of the pure counter abstraction method (rather than more general predicate or finitary abstraction) to concurrent reactive systems where:

- interaction between any processes present in the system occurs through communication of events (the node processes are interleaved so they share no events between themselves; however, there is shared communication between these processes and the rest of the system as well as the environment);
- the specification is provided as a process definition (as opposed to a formula of temporal logic) and is verified through a refinement check;
- *failures* are used in system analysis in order to allow verification of specifications that talk not only about safety of the implementation (i.e. what the system can do), but also about the availability of events (i.e. what the system cannot refuse to do); and
- the node processes are put in a general context.

Even though counter abstraction has already been applied to some extent to systems with processes interacting by communication [2,18], we are not aware of any application of the method where model checking occurs via a refinement check. Refinement checking allows for a different range of properties to be verified compared to verification using temporal formulae. Additionally, our system needs to counter abstract only the implementation process: the specification side remains unchanged.

We choose our language to be the process algebra CSP with two semantic models, *traces* and *stable failures*. We give a brief overview of the syntax and semantics of CSP and the idea of refinement in the next section; more details can be found in [9,16]. For simplicity, we assume that all the processes in our system are non-divergent and do not contain termination events.

Our overall goal, given a CSP specification process $Spec$ and a CSP implementation process $System(N) = F(Nodes(N))$, is to verify whether $Spec \sqsubseteq System(N)$ for all values of N. The model checker FDR [15,5] can perform this refinement check for small, fixed N; however, it is unable to do so when N is large and clearly it cannot help much with proving the refinement *for all* N. This is where we will use the full potential of counter abstraction: we construct a process $Abstr$ such that $Abstr \sqsubseteq Nodes(N)$ by construction, and hence $F(Abstr) \sqsubseteq F(Nodes(N)) = System(N)$; then the refinement check $Spec \sqsubseteq F(Abstr)$ can be easily performed by FDR.

The rest of the paper is structured as follows. In the next section we give a brief overview of CSP. In Section 3 we describe the idea of counter abstraction with unbounded integer counters, and introduce our running example. It is observed that, even though this may dramatically reduce the number of states in the state machine representation of the system, the state space may be still unbounded in size. We prove that such a modified system is bisimilar, and hence traces and stable-failures equivalent, to the original one. The abstraction is improved further by adding thresholds to the counters in Section 4. We show that this operation creates an anti-refinement of the original system, which is independent of the number of node processes. This allows us to perform a single refinement check against *Spec* in order to conclude that *System*(N) refines *Spec* for all N. We also briefly talk about a tool that automates the process of creating counter abstraction models, suitable for checking using FDR. In addition, we present time comparison results between explicit model checking and model checking using counter abstraction with thresholds, based on the example used throughout the paper. We conclude the paper in Section 5 with a short discussion of possible directions of further development of the method.

2 Introduction to CSP

2.1 Syntax

The CSP process algebra was introduced by Hoare in 1978 [8] and then modified into its current form in 1985 [9]. Roscoe's book [16] is probably the most complete description of the language.

For any process we define its alphabet, Σ, to be the set of all observable communication events that this process can engage in, either with the environment or with other processes. There are also a special event: the internal event τ, which a process can communicate invisibly, without any interaction from the environment. We write Σ^τ to mean $\Sigma \cup \{\tau\}$.

The partial CSP syntax that we use in this paper is the following.

$$P ::= STOP \mid a \to P \mid P \,\square\, P \mid P \sqcap P \mid P[\![{}^{b}/_{a}]\!] \mid b \,\&\, P \mid P \underset{X}{\parallel} P \mid P \,|||\, P.$$

The process *STOP* is a synonym of deadlock, i.e. it is the process that cannot engage in any communication with the environment. $a \to P$ is a process that can initially communicate the event a and then behave like process P. For two processes P and Q, $P \,\square\, Q$ (*external choice*) is a process that offers the environment the choice of performing any initial event of P or Q: if an initial event of P is performed, then the choice is resolved to P, and if an initial event of Q is performed, then the choice is resolved to Q. $P \sqcap Q$, on the other hand, represents an *internal* or *nondeterministic choice*, where the process behaves either like P or like Q: the choice is made by some mechanism that we do not model, and cannot be influenced by the environment.

$P[\![{}^{b}/_{a}]\!]$ (*renaming*) is a process that behaves like P except that whenever the process would normally perform a, it now performs b.

The processes $b \,\&\, P$ is equivalent to if b then P else $STOP$: P is enabled only if the guard b is true.

The notion of parallel composition of processes plays an important role in CSP. The process $P \underset{X}{\parallel} Q$ is the parallel composition of P and Q, with handshaken synchronisation on all the events from the set X. We also write $\big\|_{s \in S}[A(s)]P(s)$ to mean the parallel composition of processes $P(s)$ indexed over S, where each $P(s)$ is only allowed to perform events from $A(s)$ and the processes synchronise on common events. $P \,|||\, Q$ is the interleaving of P and Q: the process are in parallel, but do not synchronise on any event.

It is important to note that here we have presented only a subset of the CSP syntax. However, since all our results are derived using operational semantics, they all apply to the full CSP language provided the processes are non-divergent and contain no termination events. The syntax presented in this section is only used for our running example.

2.2 Semantics

In this paper we use the standard operational semantics for CSP (as defined in [16, chapter 7]). Given two processes P and Q and an event e, we write $P \xrightarrow{e} Q$ to mean that P can perform an event e and then behave like Q. Also we write $P \overset{tr}{\Longrightarrow} Q$ to mean that the process P can perform the trace tr of visible events (and possibly some number of τ events) and then behave like process Q.

In this paper we consider CSP processes in two different denotational semantic models: *traces* and *stable failures*. In the former each process P is described by its set of traces (written $traces(P)$), which contains all the finite sequences of events that P can communicate to the environment. In the *stable failures* model a process P is represented by the set of its traces (defined as above) together with the set of its *failures* (written $failures(P)$). A *failure* is a pair (s, X), which represents that the process can perform the trace s to reach a stable state (i.e. where no internal events are possible), where it can refuse the whole of X (i.e. none of the events of X is available).

Central to the idea of model checking using CSP is the concept of refinement. In the traces model, we say that a process P is *refined by* Q (or that Q *refines* P) if whenever Q can perform a trace tr, then so can P. Formally

$$P \sqsubseteq_T Q \Leftrightarrow traces(Q) \subseteq traces(P).$$

In the failures model, we also require the failures of Q to be in the failures of P:

$$P \sqsubseteq_F Q \Leftrightarrow traces(Q) \subseteq traces(P) \wedge failures(Q) \subseteq failures(P).$$

FDR can test whether such refinements hold, for finite state processes.

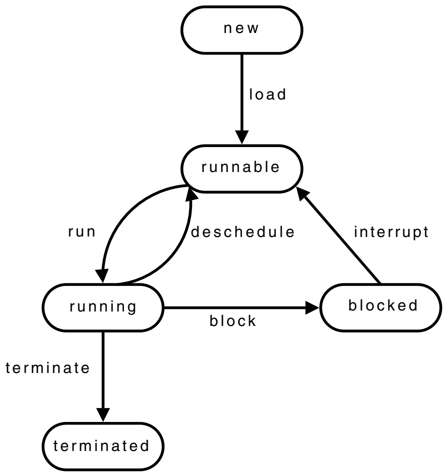

Fig. 1. State diagram of the basic model for processes within an operating system

3 Counting the processes

In this section we present a transformation method, which generates a system that is bisimilar to $Nodes(N)$. Each state in the new system is a tuple of integer counters, each counting how many node processes are in a given concrete state. We start by introducing our running example, and then describe the transformation itself.

3.1 Multiprocessor operating system example

We will model a process scheduling mechanism in an operating system for a multiprocessor machine. Our implementation will consist of a number of nodes, each representing a single operating system process requesting CPU access, and a scheduler.

We adopt the common model of process states [17], as shown in Figure 1. We model a node as below.

$$\begin{aligned}
Node(new) &= load \to Node(runnable),\\
Node(runnable) &= run \to Node(running),\\
Node(running) &= deschedule \to Node(runnable)\\
&\quad \Box\ block \to Node(blocked)\\
&\quad \Box\ terminate \to STOP,\\
Node(blocked) &= interrupt \to Node(runnable).
\end{aligned}$$

The task of the scheduler is to divide the CPU time between the different processes. Let *cores* be the number of processors present in the system. Then the scheduler, below, consists of *cores* interleaved processes $Core$, each representing a single processor resource. We abstract away the details of the algorithm that the scheduler uses to decide which process should be given access to the CPU next (letting it pick any available process nondeterministically), and hence our analysis holds for *all* scheduling algorithms.

$$Scheduler = \mathop{|||}_{i=0}^{cores-1} Core(idle)$$

The process *Core*, below, models a single CPU resource. When *Core* is idle it can *run* and become busy; and when busy, any of the events that imply that a process no longer needs CPU time (i.e. *deschedule*, *block* or *terminate*) brings it back to the idle state.

$$\begin{aligned} Core(idle) &= run \rightarrow Core(busy) \\ Core(busy) &= deschedule \rightarrow Core(idle) \\ &\quad \square\ block \rightarrow Core(idle) \\ &\quad \square\ terminate \rightarrow Core(idle). \end{aligned}$$

To create our implementation, we interleave all the node processes and put them in parallel with the scheduler process, synchronising on the common events. Finally we rename all the events that imply that a process no longer needs CPU time (i.e. *deschedule*, *block* or *terminate*) to a single event *stopRun* since, for specification verification purposes, we need not to distinguish *why* a process no longer needs a CPU.

$$\begin{aligned} Nodes(N) &= \mathop{|||}_{i=0}^{N-1} Node(new), \\ System(N) &= (Nodes(N) \underset{Alpha}{\|} Scheduler) \\ &\qquad [\![{}^{stopRun, stopRun, stopRun} / {}_{deschedule, block, terminate}]\!], \\ Alpha &= \{\!| run, deschedule, block, terminate |\!\}. \end{aligned}$$

Finally we would like our specification to say that we never have more than *cores* processes in the *running* state, i.e. the number of *run* events minus the number of *stopRun* events is never more than *cores*. Further, if there is at least one process running, then it can stop running, so the event *stopRun* is not refused. Finally, the events *load* and *interrupt* may or may not be available. Hence we define

$$\begin{aligned} Spec &= Spec'(0), \\ Spec'(x) &= x < cores \ \&\ (run \rightarrow Spec'(x+1) \sqcap STOP) \\ &\quad \square\ x > 0 \ \&\ stopRun \rightarrow Spec'(x-1) \\ &\quad \square\ (load \rightarrow Spec'(x) \sqcap interrupt \rightarrow Spec'(x) \sqcap STOP). \end{aligned}$$

We fix the value of *cores*. Our verification problem is then:

$$\forall N \geq 1 \bullet Spec \sqsubseteq_F System(N).$$

3.2 Counter abstraction with unbounded counters

Counter abstraction works by transforming an (unbounded) concrete state machine into a finite abstract state machine. The CSP language is equipped with full operational semantics [16, chapter 7], which allows us to convert a definition of any CSP process into an explicit state machine. We will represent each such state machine by a tuple

$$(S, s_0, \Sigma^\tau, \{\xrightarrow{e} \mid e \in \Sigma^\tau\}),$$

where: S is the set of states that the process can be in; s_0 is the initial state; Σ^τ is the set of events the process is allowed to communicate, extended with the internal action τ; and $\xrightarrow{e} \subseteq S \times S$ is the transition relation specifying the e-successors of each state.

We represent a single node process $Node$ as the state machine

$$\mathcal{S}_{Node} = (S_{Node}, n_0, \Sigma^\tau_{Node}, \{\xrightarrow{e} \mid e \in \Sigma^\tau_{Node}\}),$$

where $S_{Node} = \{n_0, n_1, \ldots, n_{k-1}\}$ (so that the number of states each of the node processes can be in is k).

Each state of the state machine for $Nodes(N)$ can be represented as a tuple of the form $(n_{j_0}, n_{j_1}, \ldots, n_{j_{N-1}})$, which corresponds to the process $\bigg|\bigg|\bigg|_{i=0}^{N-1} n_{j_i}$ (i.e. the i-th node process is in the state n_{j_i}).

We define an abstraction method that, using only $\mathcal{S}_{Node}$, automatically generates an abstract state machine

$$\mathcal{A}_\infty = (A_\infty, a_0, \Sigma^\tau_A, \{\xrightarrow{e}_{\mathcal{A}_\infty} \mid e \in \Sigma^\tau_A\})$$

that is bisimilar to $Nodes(N)$. Each a in A_∞ is a k-tuple of integer counters

$$(c_0, c_1, c_2, \ldots, c_{k-1}),$$

which corresponds to a concrete state

$$(n_{j_0}, n_{j_1}, \ldots, n_{j_{N-1}})$$

where each c_j counts the number of nodes in state n_j, i.e.

$$\forall\, j \in [0..k) \bullet c_j = \#\{i \in [0..N) \mid n_{j_i} = n_j\}.$$

The abstract initial state a_0 is then the tuple $(N, 0, \ldots, 0)$. This naturally corresponds to the situation where all the node processes are in their initial states.

To define the alphabet of the abstract machine, Σ^τ_A it is enough to observe that the abstract machine can only perform events that a node process could. Hence

$$\Sigma^\tau_A = \Sigma^\tau_{Node}.$$

Finally, we define the abstract transition relation $\xrightarrow{e}_{\mathcal{A}_\infty}$. Let $e \in \Sigma^\tau_A$ be given. Then

$$\begin{aligned}
&(c_0, c_1, \ldots, c_{k-1}) \xrightarrow{e}_{\mathcal{A}_\infty} (c'_0, c'_1, \ldots, c'_{k-1})\\
&\Leftrightarrow\\
&\exists\, i, j \in [0..k) \bullet n_i \xrightarrow{e} n_j \wedge c_i > 0 \wedge\\
&\qquad ((i \neq j \wedge c'_i = c_i - 1 \wedge c'_j = c_j + 1 \wedge \forall\, h \neq i, j \bullet c'_h = c_h)\\
&\qquad \vee (i = j \wedge \forall\, h \bullet c'_h = c_h)).
\end{aligned}$$

This is to say that there is an e-transition between two states in the abstract state machine if and only if there is an e-transition of some node process from state n_i

to state n_j, with $c_i > 0$ (so at least one node process is in state n_i); and the corresponding counters in the abstract state changed correctly: if n_i and n_j are different states then counter i is decreased by 1, counter j increased by 1 and all the other counters remained unchanged; if n_i and n_j are the same state then all the counters remain unchanged.

For the rest of the paper, we let $CANodes_\infty(N)$ be a process whose state machine is the abstract state machine $\mathcal{A}_\infty$ generated from $\mathcal{S}_{Node}$ as above.

Proposition 3.1 *$CANodes_\infty(N)$ and $Nodes(N)$ are strongly bisimilar.*

Proof: These two systems are strongly bisimilar by construction, with bisimulation relation:

$$\left\{\left(|||_{i=0}^{N-1} n_{j_i},\ (c_0, c_1, \ldots, c_{k-1})\right) \middle| \forall j \in [0..k) \bullet c_j = \#\{i \in [0..N) \mid n_{j_i} = n_j\}\right\}.$$

□

Corollary 3.2 *$CANodes_\infty(N) \equiv_F Nodes(N)$.*

4 Counter abstraction with thresholds

In the previous section we showed how to create an abstraction of a concurrent system by using *unbounded* integer counters. Even though such an abstraction offers a dramatic decrease in the number of states by factoring the states with respect to bisimulation, it is likely that the state space will still be unbounded in size. More importantly, its size will still depend on N. In this section we present an improved abstraction method, where we introduce a *threshold z* for the values each of the counters can take.

4.1 Constructing the state machine

Let z be a positive integer. Our aim is to create the abstract model

$$\mathcal{A}_z = (A_z, a_0, \Sigma^\tau_{A_z}, \{\xrightarrow{e}_{\mathcal{A}_z} \mid e \in \Sigma^\tau_{A_z}\})$$

by defining its set of states, the initial state, the alphabet and the transition relations.

For each state $(c_0, c_1, \ldots, c_{k-1}) \in A_\infty$ we define a state $a \in A_z$ such that

$$a = (z \sqcap c_0, z \sqcap c_1, \ldots, z \sqcap c_{k-1}),$$

where $x \sqcap y = min(x, y)$. The purpose of the variable z is to put an upper bound on the values that each of the integer counters can attain. This allows us to count the processes in the domain $\{0, 1, 2, \ldots, z-1, z \text{ or more}\}$.

The addition of the threshold z requires a new definition for the initial state. Throughout this section, assume $N \geq z$ (the value of z is usually small, so all the refinement checks for $N < z$ can be performed directly). Then

$$a_0 = (z, 0, \ldots, 0).$$

The alphabet of the abstract state machine stays unchanged. Hence

$$\Sigma^\tau_{A_z} = \Sigma^\tau_A.$$

Suppose $s_1 = (c_0, c_1, \ldots, c_{k-1})$ and $s_2 = (d_0, d_1, \ldots, d_{k-1})$ are two states in A_∞ and A_z, respectively. Then we say that s_1 and s_2 are *corresponding*, written $cor(s1, s2)$, if for all $j \in [0..k)$ we have $d_j = z \sqcap c_j$.

Now, let $e \in \Sigma^{\tau}_{A_z}$ be given. Intuitively, we want two states in $\mathcal{A}_z$ to be related by $\xrightarrow{e}_{\mathcal{A}_z}$ if and only if there exist two corresponding states in $\mathcal{A}_\infty$ related by $\xrightarrow{e}_{\mathcal{A}_\infty}$. Formally we define

$$d \xrightarrow{e}_{\mathcal{A}_z} d' \Leftrightarrow \exists\, c, c' \in A_\infty \bullet cor(c, d) \wedge cor(c', d') \wedge c \xrightarrow{e}_{\mathcal{A}_\infty} c'.$$

For the rest of the paper, we let $CANodes_z$ be a process whose state machine is the abstract state machine $\mathcal{A}_z$ generated as above.

We can easily generate $CANodes_z$ for our running example. The counter abstraction will contain five counters, one for each of the states that a single node process can be in. Below are the first two branches of the definition; the other branches are similar and can be found in Appendix A.

$$CANodes_z = CANodes'_z(z, 0, 0, 0, 0)$$

```
$CANodes'_z(c_0, c_1, c_2, c_3, c_4) =$
  $c_0 > 0$ & $load \rightarrow$ (if $c_0 = z$
     then $CANodes'_z(c_0 - 1, z \sqcap (c_1 + 1), c_2, c_3, c_4)$
          $\sqcap$ $CANodes'_z(c_0, z \sqcap (c_1 + 1), c_2, c_3, c_4)$
     else $CANodes'_z(c_0 - 1, z \sqcap (c_1 + 1), c_2, c_3, c_4)$)
  $\Box$
  $c_1 > 0$ & $run \rightarrow$ (if $c_1 = z$
     then $CANodes'_z(c_0, c_1 - 1, z \sqcap (c_2 + 1), c_3, c_4)$
          $\sqcap$ $CANodes'_z(c_0, c_1, z \sqcap (c_2 + 1), c_3, c_4)$
     else $CANodes'_z(c_0, c_1 - 1, z \sqcap (c_2 + 1), c_3, c_4)$)
  $\Box$
  ...
```

The nondeterminism present above comes from the fact that whenever one of the counters c_j is equal to z, it can mean there are either exactly z or strictly more than z node processes in state n_j.

It is intuitive to expect that counter abstracting a process using the threshold creates an anti-refinement of a counter abstraction of the same process without the threshold. The rest of the section proves this formally.

Lemma 4.1 *Suppose that*

$$CANodes_\infty(N) \overset{tr}{\Longrightarrow} (c_0, c_1, \ldots, c_{k-1}).$$

Then

$$CANodes_z \overset{tr}{\Longrightarrow} (z \sqcap c_0, z \sqcap c_1, \ldots, z \sqcap c_{k-1}).$$

Proof: This follows from the definition of $CANodes_z$, by induction on the number of transitions corresponding to tr. □

Lemma 4.2 *Let $z \geq 1$ and suppose that $\forall\, j \in [0..k) \bullet z \sqcap c_j = z \sqcap c'_j$. Then for all events e:*

$$(c_0, c_1, \ldots, c_{k-1}) \xrightarrow{e}_{\mathcal{A}_\infty} \Leftrightarrow (c'_0, c'_1, \ldots, c'_{k-1}) \xrightarrow{e}_{\mathcal{A}_\infty} .$$

Proof: From each state, the event e is available if and only if there is at least one node process in a state n_j where it can perform e (since there is no synchronisation between the nodes). This will be true if, respectively, $c_j > 0$ or $c'_j > 0$, for some such j. However, $z \geq 1$ and $\forall\, j \in [0..k) \bullet z \sqcap c_j = z \sqcap c'_i$, so $\forall\, j \in [0..k) \bullet c_j > 0 \Leftrightarrow c'_j > 0$. Hence e is available from the two states under the same circumstances. □

Proposition 4.3 $CANodes_z \sqsubseteq_F CANodes_\infty(N)$.

Proof: By Lemma 4.1 we have that $traces(CANodes_\infty(N)) \subseteq traces(CANodes_z)$.

Let $(tr, X) \in failures(CANodes_\infty(N))$. Then for some $(c_0, c_1, \ldots, c_{k-1})$, $CANodes_\infty(N) \overset{tr}{\Longrightarrow} (c_0, c_1, \ldots, c_{k-1})$ and $(c_0, c_1, \ldots, c_{k-1})$ refuses X. Then for each $e \in X \cup \{\tau\}$, $(c_0, c_1, \ldots, c_{k-1}) \not\xrightarrow{e}_{\mathcal{A}_\infty}$.

By Lemma 4.1 we have that $CANodes_z \overset{tr}{\Longrightarrow} (z \sqcap c_0, z \sqcap c_1, \ldots, z \sqcap c_{k-1})$. Let $c'_0, c'_1, \ldots, c'_{k-1}$ be such that $z \sqcap c_i = z \sqcap c'_i$ for all i. Then by Lemma 4.2, $(c'_0, c'_1, \ldots, c'_{k-1}) \not\xrightarrow{e}_{\mathcal{A}_\infty}$, for each $e \in X \cup \{\tau\}$. Hence $(z \sqcap c_0, z \sqcap c_1, \ldots, z \sqcap c_{k-1}) \not\xrightarrow{e}_{\mathcal{A}_z}$, by definition of the transition relation in $\mathcal{A}_z$, for each $e \in X \cup \{\tau\}$. Therefore $(tr, X) \in failures(CANodes_z)$. □

Corollary 4.4 $CANodes_z \sqsubseteq_F Nodes(N)$.

Proof: This follows from Corollary 3.2, Proposition 4.3 and transitivity of refinement. □

Theorem 4.5 *If $F(\_)$ is any CSP context, and Spec is any process, then*

$$Spec \sqsubseteq F(CANodes_z) \;\Rightarrow\; \forall\, N \geq z \bullet Spec \sqsubseteq F(Nodes(N)),$$

where the refinements are either both in the traces or both in the failures model.

Proof: By Corollary 4.4 and monotonicity of the CSP operators, we have $F(CANodes_z) \sqsubseteq_F F(Nodes(N))$, and hence $F(CANodes_z) \sqsubseteq_T F(Nodes(N))$, for all $N \geq z$. The result then follows from transitivity of refinement. □

4.2 Tool support

We have created a simple automated tool called TomCAT[3] that, given an appropriate CSP script and the value of z, automatically produces the CSP definition

[3] Available from `http://web.comlab.ox.ac.uk/people/tomasz.mazur/`.

of the counter-abstracted process $CANodes_z$. Most of the parsing work is done by FDR: given a CSP script, it produces the concrete state machine of a single node process. Then the tool produces an output CSP script identical to the original one with the exception that the original definition of the process $Nodes$ is replaced by its counter-abstracted version.

The definition from Section 4.1 is inefficient to compile in FDR, as it uses a single sequential process with $(z+1)^k$ states. TomCAT produces a CSP-equivalent, but more efficient definition, using a parallel composition of k separate processes, one for each counter, giving a total of $k(z+1)$ states to compile.

The CSP script produced by TomCAT for the running example is below. Here $States$ and $Transitions$ define the set of states and transitions of the abstract state machine, respectively. Given a state s, $incoming(s)$, $selfloops(s)$ and $outgoing(s)$ are sets of transitions for which s is the end state only, both the start and the end state and the start state only, respectively. Each member of these sets is defined to be a tuple $(s0, e, s1)$ and represents the corresponding transition $s0 \xrightarrow{e} s1$. $Counter(s, x)$ represents a single counter for the state s storing its value in x. Further, $Counters$ is the parallel composition of all the counters; the alphabets $alpha(s)$, are chosen to ensure that each counter $Counter(s, x)$ participates in those events coresponding to state s. Finally, $CANodes_z$ is the process whose state machine is the abstract state machine generated from the process $Node$ using the counter abstraction method with threshold (Section 4.1). It is obtained by hiding the events $TAU\_$ that represent internal actions in order to achieve the true τ's, and then replacing the events representing other transitions by the labels of these transitions.

Most of the definitions below are common for all outputs. The only parts that change are the definitions of the threshold z (supplied by the user on the command line) and $States$ and $Transitions$ (both of which are generated by FDR).

$$z = cores + 1$$

$$States = \{0..4\}$$

$$initState = 0$$

$$\begin{aligned} Transitions = \{&(0, load, 1), (1, run, 2), (2, terminate, 4), (2, deschedule, 1),\\ &(2, block, 3), (3, interrupt, 1)\} \end{aligned}$$

$$incoming(s) = \{(s0, e, s1) \mid (s0, e, s1) \in Transitions, s0 \neq s \wedge s1 = s\}$$

$$selfloops(s) = \{(s0, e, s1) \mid (s0, e, s1) \in Transitions, s0 = s \wedge s1 = s\}$$

$$outgoing(s) = \{(s0, e, s1) \mid (s0, e, s1) \in Transitions, s0 = s \wedge s1 \neq s\}$$

$$
\begin{aligned}
&Counter(s,x) = \\
&\quad do?t : incoming(s) \to Counter(s, min(x+1, z)) \\
&\quad \Box\ x > 0\ \&\ do?t : selfloops(s) \to Counter(s,x) \\
&\quad \Box\ x > 0\ \&\ do?t : outgoing(s) \to \\
&\qquad \text{if } x = z \text{ then } Counter(s,x) \sqcap Counter(s, x-1) \\
&\qquad\qquad \text{else } Counter(s, x-1)
\end{aligned}
$$

$$
\begin{aligned}
&alpha(s) = \{do.t \mid t \in \bigcup\{incoming(s), selfloops(s), outgoing(s)\}\} \\
&Counters = \big\|_{s \in States} [alpha(s)]\, Counter(s,\ \text{if } s = initState \text{ then } z \text{ else } 0) \\
&CANodes_z = \\
&\quad (Counters \setminus \{do.(s0,e,s1) \mid (s0,e,s1) \in Transitions, e = TAU\_\}) \\
&\qquad [\![{}^{e}/_{do.(s0,e,s1)} \mid (s0,e,s1) \in Transitions, e \neq TAU\_]\!]
\end{aligned}
$$

4.3 Experimental results

Going back to our example, let

$$F(P) = (P \underset{Alpha}{\|} Scheduler)[\![{}^{stopRun,stopRun,stopRun}/_{deschedule,block,terminate}]\!],$$

with *Alpha* as before. We can use FDR to verify that

$$Spec \sqsubseteq F(CANodes_z).$$

In this case we need to take z to be at least $cores + 1$ for the refinement to hold. Then, by Theorem 4.5, we can deduce

$$\forall N \geq z \bullet Spec \sqsubseteq System(N).$$

Table 1 shows the time it takes to model check the concrete and abstract state machines for our example, with $cores = 1, 2, \ldots, 5$ and $N = 5, 6, \ldots, 10$. FDR completed all the checks with counter abstraction models in less than 1 second.

The number of states for the concrete state machine grows exponentially[4] in N. The number of abstract states, on the other hand, remains constant for all values of N.

More generally, the abstraction of the nodes has $O((z+1)^k)$ states, which is exponential in the size of the concrete state machine of a single node process; however, in most practical situation many of these states are unreachable so the typical case is much better.

[4] The exact number of states that FDR checks is in fact $\sum_{k=0}^{cores} \binom{cores}{k} \binom{N}{k} 4^{N-k}$.

	$cores = 1$	$cores = 2$	$cores = 3$	$cores = 4$	$cores = 5$
$N = 5$	< 1	< 1	< 1	< 1	1
$N = 6$	< 1	< 1	1	3	5
$N = 7$	1	3	6	12	22
$N = 8$	5	13	28	63	123
$N = 9$	18	61	167	371	806
$N = 10$	94	336	1026	1978	4368

Table 1
Time comparison (in seconds) for explicit model checking.

5 Future work

An important question to ask is what value to choose for the threshold z. If it is too low, then if the refinement $Spec \sqsubseteq CANodes_z$ does not hold, it is likely that we get a spurious counterexample that does not occur in the concrete system. We can automate the processes of finding the smallest possible z (if one exists) for which our final refinement check holds, or produces a counterexample that occurs in the original system, using techniques similar to counterexample-guided abstraction refinement [3], as below:

Step 1: Let $z = 1$.
Step 2: If $Spec \sqsubseteq F(CANodes_z)$, then done.
Step 3: Else check if the counterexample for the refinement is also a counterexample for $Spec \sqsubseteq F(Nodes(N))$.
Step 4: If yes then conclude that $Spec \not\sqsubseteq F(Nodes(N))$.
Step 5: Else let $z = z + 1$ and go to Step 2.

This procedure is not guaranteed to terminate, but if it does, then it gives us the z we want.

Another possibility is to replace the single threshold z by a tuple of thresholds, one for each of the integer counters. In this way each of the counters could have a different cut-off value.

Another possible area of improvement is the family of systems allowed for the application of our method. At the moment we do not allow the node processes to use the node identifiers inside their definitions, for otherwise the state space of the node processes would be dependent on N. This limits the number of systems that could be analysed using counter abstraction and FDR. We intend to address this problem using techniques from data independence [11,16] to collapse, within each node process, the type $[0..N)$ of node identifiers to some fixed finite type T, and then to proceed using the methods described in this paper.

Acknowledgement

We would like to thank Bill Roscoe, Michael Goldsmith, Joël Ouaknine and the anonymous referees for helpful comments on this work. The research was partially supported by a grant from the EPSRC.

References

[1] Apt, K.R., and D.C. Kozen, *Limits for automatic verification of finite-state concurrent systems*, Information Processing Letters, **22(6)** (1986), 307–309.

[2] Clarke, Edmund M., and Orna Grumberg, *Avoiding the state explosion problem in temporal logic model checking*, PODC '87: Proceedings of the Sixth Annual ACM Symposium on Principles of Distributed Computing, ACM Press, New York (1987), 294–303.

[3] Clarke, Edmund M., Orna Grumberg, Somesh Jha, Yuan Lu, and Helmut Veith, *Counterexample-guided abstraction refinement*, CAV'00, Springer-Verlag, London (2000), 154–169.

[4] Das, Satyaki, *Predicate abstraction*, Ph.D. thesis, Stanford University, 2003.

[5] Formal Systems (Europe) Ltd., *Failures-Divergence Refinement — FDR 2 user manual*, 1999, URL: http://www.fsel.com/fdr2_manual.html.

[6] Graf, Susanne, and Hassen Saïdi, *Construction of abstract state graphs with PVS*, CAV'97, Springer-Verlag, London (1997), 72–83.

[7] German, Steven M., and A. Prasad Sistla. *Reasoning about systems with many processes*, J. ACM, **39(3)** (1992), 675–735.

[8] Hoare, C. A. R., *Communicating sequential processes*, Commun. ACM, **21(8)** (1978), 666–677.

[9] Hoare, C. A. R., "Communicating sequential processes", Prentice Hall Europe, 1985.

[10] Kesten, Yonit, and Amir Pnueli, *Verification by augmented finitary abstraction*, Information and Computation, **163(1)** (2000), 203–243.

[11] Lazić, R. S., *A semantic study of data independence with applications to model checking*, Ph.D. thesis, Oxford University Computing Laboratory, 1999.

[12] Lahiri, S., and R. Bryant, *Constructing quantified invariants via predicate abstraction*, Conference on Verification, Model Checking and Abstract Interpretation (VMCAI '04), LNCS, **2937** (2004), 267–281.

[13] Manna, Zohar, and Amir Pnueli, "Temporal verification of reactive systems: safety", Springer-Verlag, New York, 1995.

[14] Pnueli, Amir, Jessie Xu, and Lenore D. Zuck. *Liveness with (0, 1, ∞)-counter abstraction*, CAV'02, Springer-Verlag, London (2002) 107–122.

[15] Roscoe, A. W., *Model-checking CSP*, "A Classical Mind, Essays in Honour of C. A. R. Hoare", Prentice-Hall, 1994.

[16] Roscoe, A. W., "The Theory and Practice of Concurrency," Prentice Hall Series in Computer Science, Prentice-Hall, London, New York (1998). With associated web site http://www.comlab.ox.ac.uk/oucl/publications/books/concurrency/.

[17] Tanenbaum, Andrew S., *Operating systems: design and implementation*, Prentice-Hall, Upper Saddle River, 1987.

[18] Vernier, Isabelle, *Parameterized evaluation of CTL-X formulae*, Proceedings of the 1st International Conference on Temporal Logic workshop, 1994.

[19] Xu, Jessie, *Automatic verification of parameterized systems*, Ph.D. thesis, New York University, 2005.

A Full definition of $CANodes_z$ for the running example

$$
\begin{array}{l}
CANodes_z(c_0, c_1, c_2, c_3, c_4) = \\
\quad c_0 > 0 \ \& \ load \rightarrow (\text{if } c_0 = z \\
\qquad \text{then } CANodes_z(c_0 - 1, z \sqcap (c_1 + 1), c_2, c_3, c_4) \\
\qquad\qquad \sqcap\ CANodes_z(c_0, z \sqcap (c_1 + 1), c_2, c_3, c_4) \\
\qquad \text{else } CANodes_z(c_0 - 1, z \sqcap (c_1 + 1), c_2, c_3, c_4)) \\
\quad \Box \\
\quad c_1 > 0 \ \& \ run \rightarrow (\text{if } c_1 = z \\
\qquad \text{then } CANodes_z(c_0, c_1 - 1, z \sqcap (c_2 + 1), c_3, c_4) \\
\qquad\qquad \sqcap\ CANodes_z(c_0, c_1, z \sqcap (c_2 + 1), c_3, c_4) \\
\qquad \text{else } CANodes_z(c_0, c_1 - 1, z \sqcap (c_2 + 1), c_3, c_4)) \\
\quad \Box \\
\quad c_2 > 0 \ \& \ deschedule \rightarrow (\text{if } c_2 = z \\
\qquad \text{then } CANodes_z(c_0, z \sqcap (c_1 + 1), c_2 - 1, c_3, c_4) \\
\qquad\qquad \sqcap\ CANodes_z(c_0, z \sqcap (c_1 + 1), c_2, c_3, c_4) \\
\qquad \text{else } CANodes_z(c_0, z \sqcap (c_1 + 1), c_2 - 1, c_3, c_4) \\
\quad \Box \\
\quad c_2 > 0 \ \& \ block \rightarrow (\text{if } c_2 = z \\
\qquad \text{then } CANodes_z(c_0, c_1, c_2 - 1, z \sqcap (c_3 + 1), c_3) \\
\qquad\qquad \sqcap\ CANodes_z(c_0, c_1, c_2, z \sqcap (c_3 + 1), c_4) \\
\qquad \text{else } CANodes_z(c_0, c_1, c_2 - 1, z \sqcap (c_3 + 1), c_4)) \\
\quad \Box \\
\quad c_2 > 0 \ \& \ terminate \rightarrow (\text{if } c_2 = z \\
\qquad \text{then } CANodes_z(c_0, c_1, c_2 - 1, c_3, z \sqcap (c_4 + 1)) \\
\qquad\qquad \sqcap\ CANodes_z(c_0, c_1, c_2, c_3, z \sqcap (c_4 + 1)) \\
\qquad \text{else } CANodes_z(c_0, c_1, c_2 - 1, c_3, z \sqcap (c_4 + 1)) \\
\quad \Box \\
\quad c_3 > 0 \ \& \ interrupt \rightarrow (\text{if } c_3 = z \\
\qquad \text{then } CANodes_z(c_0, z \sqcap (c_1 + 1), c_2, c_3 - 1, c_4) \\
\qquad\qquad \sqcap\ CANodes_z(c_0, z \sqcap (c_1 + 1), c_2, c_3, c_4) \\
\qquad \text{else } CANodes_z(c_0, z \sqcap (c_1 + 1), c_2, c_3 - 1, c_4)).
\end{array}
$$