Formal verification of not fully symmetric systems using counter abstraction

Tomasz Mazur

Oxford University Computing Laboratory
Wolfson Building, Parks Road, Oxford OX1 3QD, UK
tomasz.mazur@comlab.ox.ac.uk

Supervisor(s): Gavin Lowe, Oxford University Computing Laboratory
Keywords: counter abstraction, parameterised verification, ring topology

Abstract. Counter abstraction allows us to transform a concurrent system with an unbounded number of agents into a finite-state bounded abstraction, independent of the number of processes present in the implementation. In its general form it is not well suited for verification of parameterised concurrent systems based on message passing that are not fully symmetric and/or use two-way handshaken synchronisation between processes. In this paper we present a method whose main idea is to count processes that are in certain equivalence classes. We use labelled transitions systems to model processes (both implementation and specification) and traces refinement for verification checks. Refinement is checked automatically using the FDR model checker. We illustrate the method on a token ring mutual exclusion algorithm from [8].

1 Introduction

There is a rapidly growing demand for verification of concurrent systems using formal methods. A particularly hard problem in this area is the *Parameterised Model Checking Problem*, a subclass of which can be described as follows:

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

It has been shown in [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.

Counter abstraction is a well-known abstraction method, probably best presented by Pnueli et al. in [6]. In general, it aims to create an abstraction of a parallel composition of N node processes, which is independent of N. Each abstract state is defined to be a tuple of integer counters. There is one counter for each concrete state that a node process can be in. Each counter is also given a non-negative threshold z which, when reached, indicates that there are z *or more* process in the corresponding state.

In [5] we showed how to build counter abstraction models for systems with unboundedly many node processes, none of which used node identifiers. However, in the majority of real world applications node processes do use their own identity or identities of other nodes within their definition. The biggest problem with this is that it makes the alphabets of processes unboundedly large. Our current research addresses this problem. In the next section we introduce the necessary notation. In Section 3 we present a brief outline of our method and state some useful results. Section 4 shows how our method can be applied in verification of a system with a ring topology using an example similar to one from [8].

2 Labelled transitions systems

We represent processes using labelled transition systems. An LTS for a node with identity me in a system with N nodes, $\mathcal{L}_{me}(N)$, is a tuple $(S, s_0, A(me, N), \longrightarrow)$, where S is a set of states, s_0 is an initial state, $A(me, N)$ is a set of transition labels and $\longrightarrow \subseteq S \times A(me, N) \times S$ is a transition relation (we write $s \xrightarrow{e} s'$ to mean $(s, e, s') \in \longrightarrow$). We assume that each process holds an identity of at most one other node. We let each state in S be of the form (st, id), where st records the control state and values of variables other than node identities, and id holds an identity of a node different from itself or is equal to $\perp$ when the node holds no such identity. We let all the labels in $A(me, N)$ be either τ (internal actions) or have one of the following forms: $c1.me.p$ (type1 labels; private events), $c2.me.other.p$ (type2a labels; two-way synchronisation events), $c2.other.me.p$ (type2b labels; two-way synchronisation events) or $c3.p$ (type3 labels; global synchronisation events), where $c1, c2$ and $c3$ are channel names, $other$ is a value in $[1..N]$ and p is some payload, independent of node identities.

We define $\mathcal{L}\ {}_X\|_Y\ \mathcal{L}'$ to be an LTS representing a parallel composition of $\mathcal{L}$ and $\mathcal{L}'$ with synchronisation on common non-τ labels (i.e. those in $(X \cap Y) \setminus \{\tau\}$). We also define $\big\|_{me=1}^{N}[A(me, N)]\mathcal{L}_{me}(N)$ to be an LTS representing a replicated parallel composition of $\mathcal{L}_1, \ldots, \mathcal{L}_N$ with any two LTSs $\mathcal{L}_i$ and $\mathcal{L}_j$ synchronising on $A(i, N) \cap A(j, N)$ (type2 labels), and all the LTSs synchronising on $\bigcap\{A(me, N) \mid me \in [1..N]\}$ (type3 labels). In addition, given a renaming function $\phi : \bigcup\{A(me, \infty) \mid me \in [1..\infty)\} \to X$ for some set of labels X, we define the ϕ-renamed LTS $\phi(\mathcal{L})$ to be exactly like $\mathcal{L}$, except every transition label e is renamed to be $\phi(e)$. Finally, given a set of labels Y, we define $\mathcal{L} \setminus Y$ to behave exactly like $\mathcal{L}$, except that every transition label e in Y is replaced with τ.

We generate all the LTSs automatically from CSP descriptions [3, 7]. The implementations we consider are of the form $\big\|_{me=1}^{N}[A(me, N))]\mathcal{L}_{me}(N)$ (possibly in parallel with a controller process and renamed), with $\mathcal{L}_i$ and $\mathcal{L}_j$ being identical up to exchanging the identities i and j. Each $\mathcal{L}_i$ is assumed to be data independent in the type of node identities (i.e. we allow variables of this type (but not constants) and ban any operations that would determine or constrain what this type is). It is also essential that there are no equality tests between variables (and constants) of the type of node identities within node process definition. Specifications are also given in a form of an LTS. We define a partial order

$\sqsubseteq$ on the set of all LTSs to be the standard trace-refinement order (as defined in [7, chapter 1]). We say that an implementation $Impl$ satisfies a specification $Spec$ if and only if $Spec \sqsubseteq Impl$.

3 Method overview

In standard counter abstraction, there is a single counter for every state in the concrete state space S, which counts how many processes are currently in this state. However, this prohibits processes to use node identifiers in their definition. With an unbounded type of node identities, S is unbounded and therefore so is the number of counters. One of the novel ideas of our method is to use an equivalence relation $\approx$, defined on S, such that if (st, id) and (st', id') are two states in S, then $(st, v) \approx (st', v')$ if and only if $st = st'$. We write $[st]$ to mean the equivalence class of a state (st, id).

Due to lack of space we are unable to present all the details of our method of creating counter abstraction, but the general idea is the following. For a fixed N, let ϕ be a renaming which renames all node identities from $[1..N]$ to 1, i.e. $\phi(\tau) = \tau$, $\phi(i) = 1$ for all i in $[1..N]$ and $\phi(i) = i$ for all i not in $[1..N]$. We lift ϕ to apply to more complex labels in the natural way (e.g. $\phi(c1.2.p) = c1.1.p$ and $\phi(c2.1.3.p) = c2.1.1.p$). We let each abstract state be a k-long vector $count$, where k is the number of equivalence classes under $\approx$ and $count(st)$ is the counter corresponding to $[st]$. The abstract initial state is defined to be a vector with all the counters equal to 0, except for the one corresponding to the concrete initial state, s_0, whose value is N. We define an abstract transition relation as follows. For every e being a type1 label or τ we have a corresponding abstract transition with label $\phi(e)$ if there is a concrete state (st, id) that can do e, the counter corresponding to $[st]$ is at least 1, and if the corresponding counters in the abstract state are changed correctly. For $e = c2.i.j.p$ (a type2 label) with $i \neq j$ we have a corresponding abstract transition with the label $\phi(e)$ if there are concrete states (st_1, id_1) and (st_2, id_2), which can do $c2.i.other.p$ and $c2.other'.j.p$, respectively, the counters corresponding to $[st_1]$ and $[st_2]$ are equal to at least 1 (2 if $st_1 = st_2$), and the corresponding counters in the abstract state are changed correctly. The case when e is a type3 label is similar, but we now require *all* processes to be in a state from which e is possible.

An abstract LTS built using the method above still depends on N. We build an improved abstraction using a method similar to the above, except that we introduce a threshold value z such that when a value of any counter is equal to z, it means that there are either z or more node processes in a corresponding concrete state, and the counter can nondeterministically decide which one it is. Let $\zeta_z(\mathcal{L}_i(x))$ be a counter abstraction of an implementation consisting of at least one node process $\mathcal{L}_i(x)$, each of which has the set of transition labels based on there being x nodes, and with threshold value z. We can show that by using counter abstraction we create an anti-refinement of the ϕ-renamed implementation. The following lemma captures this formally.

Lemma 1. *If* $N \geq 1$*, then* $\forall j \in [1..N] \bullet \zeta_z\left(\mathcal{L}_j(N)\right) \sqsubseteq \phi\left(\|_{i=1}^{N}[A(i,N)]\mathcal{L}_i(N)\right)$.

Lemma 2 says that if we fix z, then a counter abstraction of a system consisting of processes with sets of transition labels based on there being 2 nodes is equivalent to a counter abstraction of an implementation consisting of processes with sets of transition labels based on there being more than 2 processes.

Lemma 2. *If $N \geq max\{z, 2\}$ and $i \in \{1, 2\}$, then $\zeta_z(\mathcal{L}_i(2)) \equiv \zeta_z(\mathcal{L}_i(N))$.*

The following result follows from Lemma 1 and Lemma 2. Observe that ϕ is applied *after* the nodes are composed in parallel; in particular it does not remove any behaviour that we achieve by allowing the processes to use node identifiers.

Proposition 1. *If $N \geq max\{z, 2\}$, then $\zeta_z(\mathcal{L}_1(2)) \sqsubseteq \phi\left(\|_{i=1}^{N}[A(i,N)]\mathcal{L}_i(N)\right)$.*

Our main result, below, allows us to perform a refinement check of a finite and bounded abstraction of an implementation with unboundedly many node processes against a finite and bounded specification and conclude that the implementation satisfies the original specification, regardless of the value of N. For this we need some definitions. Given a set X, we say that ϕ satisfies **NoNewSharSync**$_X$ if for all non-τ labels e we have that $\phi(e) \in \phi(X) \Rightarrow e \in X$. Similarly, given sets X and Y, we say that ϕ satisfies **NoNewAlphSync**$_{X,Y}$ if for all non-τ labels e we have that $\phi(e) \in \phi(X) \cap \phi(Y) \Rightarrow e \in X \cap Y$. A data independent process satisfies **PosConjEqT** if whenever a failure of an equality test between node identifiers (and possibly constants from $[1..N]$) leads to a state with no transitions (the formal definition can be found in [4]).

Theorem 1. *Let $Ctrl(N)$ be a controller process (with a set of transition labels A_c). Suppose $Spec(N)$ and ϕ are such that if tr is a trace of $Spec(N)$ and $\phi(tr) = \phi(tr')$, then tr' is also a trace of $Spec(N)$. Also suppose that $N \geq max(z, 2)$, ϕ satisfies **NoNewAlphSync**$_{A_c, \bigcup\{A(i,N)\ \mid\ i \in [1..N]\}}$ and **NoNewSharSync**$_X$, $Spec(N)$ satisfies **PosConjEqT** and $Ctrl(1) \sqsubseteq \phi(Ctrl(N))$.*

Then if $Spec(1) \sqsubseteq \left(Ctrl(1)\ {}_{\phi(A_c)}\|_{\phi(A(1,2))}\ \zeta_z(\mathcal{L}_1(2))\right) \setminus \phi(X)$,
then $Spec(N) \sqsubseteq \left(Ctrl(N)\ {}_{A_c}\|_{\bigcup\{A(i,N)\ \mid\ i \in [1..N]\}}\ \|_{i=1}^{N}[A(i,N)]\mathcal{L}_i(N)\right) \setminus X$.

4 Ring topology example

Verification of an implementation based on a ring topology is a good example of the strengths of our method. Not only do the nodes use node identifiers in their definitions, but also each node is only allowed to communicate with its predecessor and successor. In this section we verify a distributed mutual exclusion algorithm from [8] using the techniques outlined in Section 3. Our implementation models a token ring consisting of N identical (up to identities i) node processes $Node(i)$ and a single "initialised" node processes $INode$ connected in parallel. Formally, if $X = \{token.i.j \mid i, j \in [0..N]\}$, then

$$Impl(N) = \left(INode\ {}_{A(0)}\|_{\bigcup\{A(i,N)\ \mid\ i \in [1..N]\}}\ \left(\|_{i=1}^{N}[A(i,N)]Node(i)\right)\right) \setminus X.$$

Each node process can receive a token and then nondeterministically choose either to pass the token to the successor or to perform its critical section and then pass the token. Figure 1a shows an LTS of a node process with identity i. $INode$ is identical to $Node$, except that its initial state is 1, rather than 0; this means that $INode$ is the process that initially possesses the token. We make $INode$ the predecessor of $Node(1)$ and successor of $Node(N)$.

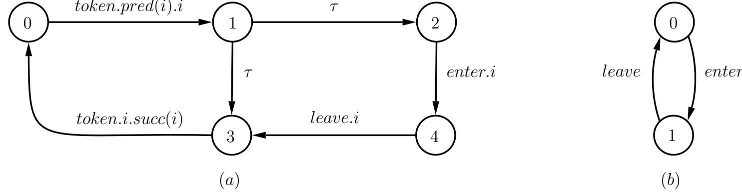

Fig. 1. Node process template LTS (a) and specification component LTS (b).

In order to satisfy the mutual exclusion property, the events *enter* and *leave* have to alternate. Hence we let *Spec* be the LTS shown in Figure 1b. If for all i we let $f(enter.i) = enter$ and $f(leave.i) = leave$, then our verification problem becomes (the case $N = 1$ can be easily verified separately):

$$\forall N \geq 2 \bullet Spec \sqsubseteq f(Impl(N)).$$

In order to create a counter abstraction of the arc of processes of $Node(1)$, $Node(2)$, ..., $Node(N)$, we need to slightly modify the LTS $Node(i)$ in order to be a better representative of a generic process $GNode(i, N)$ in the arc. When such a generic process receives a token, it could have been sent either by $INode$ (if the accepting process is the first one in the arc) or by another process $j \neq i$ within the arc (if the accepting process is any but the first process in the arc). Similarly, when it sends the token, it can address it to $INode$ (if the sending process is the last one in the arc) or to another process $j \neq i$ within the arc (if the sending process is any but the last process in the arc). We can capture this formally by saying that $GNode(i, N)$ is the same as $Node(i)$ except for the following difference: if $st \xrightarrow{e} st'$ is a transition of $Node(i)$, then $st \xrightarrow{e'} st'$ is a transition of $GNode(i, N)$ for all e' obtained from e by replacing *other* with any value that *other* can take in a corresponding transition in some node, say $Node(j)$. Figure 2 shows an LTS of such a process (in a concrete LTS any transition containing the identifier j would be replaced by multiple transitions, one for every value in $[1..N] \setminus \{i\}$).

Next, we build a counter abstraction $CAbstr = \zeta_2\,(GNode(1, 2))$. To automate this we use our prototype tool, TomCAT[1]. At this point we can use the FDR model checker [2] to verify that

$$Spec \sqsubseteq \big(INode\ {}_{\phi(A(0))}\|_{\phi(A(1,2))}\ CAbstr\big) \setminus \phi(X).$$

Observe that *Spec* and $INode$ are independent of N, so $Spec = Spec(1) = Spec(N)$ and $INode = INode(1) \sqsubseteq \phi(INode(N))$. It is not too difficult to

[1] Available from http://web.comlab.ox.ac.uk/oucl/work/tomasz.mazur/.

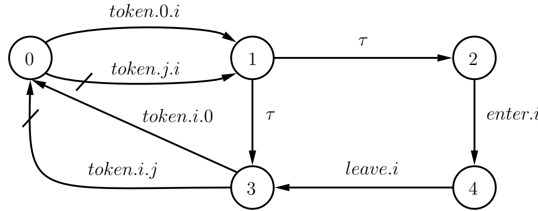

Fig. 2. An LTS of a generic arc process.

check that the conditions of Theorem 1 are satisfied. From the theorem, the fact that $GNode(i, N) \sqsubseteq Node(i)$ and monotonicity we can infer that if $N \geq 2$, then

$$f(Spec) \sqsubseteq f\left(\left(INode\ {}_{A(0)}\|_{\bigcup\{A(i,N)\,|\,i\in[1..N]\}}\left(\|_{i=1}^{N}[A(i,N)]Node(i,N)\right)\right)\setminus X\right).$$

However, $f(Spec) = Spec$, so the above solves our verification problem.

5 Future work

We currently work on extending the method to other notions of process refinement (e.g. the CSP stable failures model) to allow liveness specifications. In addition, we aim to provide improved, easier to use and more general versions of Theorem 1. We also look at larger case studies where our method is applicable. Finally, we work on the development of the TomCAT tool in order to fully automate the process of creating counter abstractions.

6 Acknowledgements

I would like to thank Gavin Lowe for many useful ideas and discussions about this paper and the anonymous referee for helpful comments on the work. The research was supported by a grant from the EPSRC.

References

1. K. R. Apt and D. C. Kozen. Limits for automatic verification of finite-state concurrent systems. *Information Processing Letters*, 22(6):307–309, 1986.
2. Formal Systems (Europe) Ltd. *Failures-Divergence Refinement-FDR 2 user manual. Available via URL* `http://www.fsel.com/fdr2_manual.html`, 1999.
3. C. A. R. Hoare. *Communicating sequential processes.* 1985.
4. R. S. Lazić. *A semantic study of data independence with applications to model checking.* PhD thesis, Oxford University Computing Laboratory, 1999.
5. T. Mazur and G. Lowe. Counter abstraction in the CSP/FDR setting. In *Proceedings of AVoCS'07*, 2007.
6. A. Pnueli, J. Xu, and L. D. Zuck. Liveness with (0, 1, ∞)-counter abstraction. In *CAV'02*, pages 107–122, 2002.
7. A. W. Roscoe. *The theory and practice of concurrency.* 1997.
8. P. Wolper and V. Lovinfosse. Verifying properties of large sets of processes with network invariants. In *Proceedings of the International Workshop on Automatic Verification Methods for Finite State Systems*, pages 68–80, 1990.