View abstraction for systems with component identities

Gavin Lowe

Department of Computer Science, University of Oxford, UK
gavin.lowe@cs.ox.ac.uk

Abstract. The *parameterised verification problem* seeks to verify all members of some family of systems. We consider the following instance: each system is composed of an arbitrary number of similar *component processes*, together with a fixed number of *server* processes; processes communicate via synchronous message passing; in particular, each component process has an *identity*, which may be included in messages, and passed to third parties. We extend Abdulla et al.'s technique of *view abstraction*, together with techniques based on symmetry reduction, to this setting. We give an algorithm and implementation that allows such systems to be verified for an arbitrary number of components. We show how this technique can be applied to a concurrent datatype built from reference-linked nodes, such as a linked list. Further, we show how to capture the specification of a queue or of a stack.

1 Introduction

The *parameterised verification problem* considers a family of systems $P(x)$ where the parameter x ranges over a potentially infinite set, and asks whether such systems are correct for all values of x. In this paper we consider the following instance of the parameterised verification problem. Each system is built from some number of similar *replicated component processes*, together with a fixed number of *server processes*; the parameter is the number of component processes. The components and servers communicate via (CSP-style) synchronous message passing; we call each message an *event*. In particular each component has an *identity*, drawn from some potentially infinite set. These identities can be included in events; thus a process can obtain the identity of a component process, and possibly pass it on to a third process. This means that each process has a potentially infinite state space (a finite control state combined with data from a potentially infinite set). We describe the setting for our work more formally in the next section. The problem is undecidable in general [5, 23]; however, verification techniques prove effective on a number of specific problems.

We adapt the technique of *view abstraction* of Abdulla et al. [1] to this setting. The idea of view abstraction is that we abstract each system state to its *views* of some size k, recording the states of just k of the replicated component processes. We can (with a finite amount of work) calculate an over-estimate of all views of size k of the system; this gives us an over-estimate of the states of the system. We then check that all states of this over-estimate satisfy our correctness condition: if so, we can deduce that *all* systems of size k or larger are correct (systems of smaller sizes can be checked directly). We present our use of view abstraction in Sections 3 and 4.

Our setting is made more complicated by the presence of the identities of the components. These mean that the set of views (for some fixed k) is potentially infinite. However, in Section 5 we use techniques from *symmetry reduction* [10, 14, 19, 9, 28, 8, 17] to reduce the views that need to be considered to a finite set.

We present the main algorithm in Section 6, and prove its correctness. We then present our prototype implementation: this is based upon the process algebra CSP [26], and builds on the model checker FDR [16], so as to support all of machine-readable CSP. We stress, though, that the main ideas of this paper are not CSP-specific: they apply to any formalism with a similar computational model, and, we believe, could be adapted to other computational models.

A major advantage of this use of identities is that it allows us to model and analyse reference-linked data structures, such as linked lists. Each node in the data structure is modelled by a replicated component process; such a process can hold the identity of another such process, modelling a reference to that node. We illustrate this technique in Section 7 by modelling and analysing a simple lock-based concurrent queue and stack, each based on a linked list; in particular, we show how to capture the specifications of these datatypes in a finite-state way, using techniques from data independence [30]. Our longer-term aim is to extend these techniques to more interesting, lock-free concurrent datatypes.

We see our main contributions as follows.

- An adaptation of view abstraction to synchronous message passing (this is mostly a straightforward adaptation of the techniques of [1]);
- An extension of view abstraction to include systems where components have identities, and these identities can be passed around, using techniques based on symmetry reduction to produce a finite-state abstraction;
- The implementation of these ideas, using FDR so as to support all of machine-readable CSP;
- The application to reference-linked concurrent datatypes;
- The finite-state specification of a queue and a stack.

1.1 Related work

There have been many approaches to the parameterised model checking problem.

Much recent work has been based on *regular model checking*, e.g. [20, 31, 7, 12]. Here, the state of each individual process is from some finite set, and each system state is considered as a word over this finite set; the set of initial states is a regular set; and the transition relation is a regular relation, normally defined by a transducer. An excellent survey is in [3]. Techniques include widening [29], acceleration [2] and abstraction [6].

The work [1] that the current paper builds on falls within this class. However, our setting is outside this class: the presence of component identities means that each individual process has a potentially infinite state space.

Other approaches include induction [13, 15, 27], network invariants [32], and counter abstraction [22, 11, 25, 23]. In particular, [23] applied counter abstraction to systems, like in the current paper, where components had identities which could be passed from one process to another: some number B of the identities were treated faithfully, and the remainder were abstracted; the approach of the current paper seems better able to capture relationships between components, as required for the analysis of reference-linked data structures.

The work [4] tackles a similar problem to this paper. It captures the specification of a queue or a stack using an automaton that, informally, guesses the data value that will be treated incorrectly. The authors use shape analysis to finitely analyse data structures built on linked lists. They are able to prove linearizability of concurrent datatypes assuming explicit linearization points are given.

Most approaches to symmetry reduction in model checking [10, 14, 19, 8, 17] work by identifying symmetric states, and, during exploration, replace each state encountered with a *representative member* of its symmetry-equivalence class: if several states map to the same representative, this reduces the work to be done. This representative might not be unique, since finding unique representatives is hard, in general; however, such approaches work well in most cases. Our approach is closer to that of [28]: we test whether a state encountered during exploration is equivalent to a state previously encountered, and if so do not explore it further.

2 The framework

In this section, we introduce more formally the class of systems that we consider, and our framework.

We introduce a toy example to illustrate the ideas. The replicated components run a simple token-based mutual exclusion protocol. Component j can receive the token from component i via a transition with event $pass.i.j$; it can then enter and leave the critical section, before passing the token to another component. In the initial state, a single component holds the token.

A *watchdog* server process observes components entering and leaving the critical section, and signals with event *error* if mutual exclusion is violated. Our correctness condition will be that the event *error* does not occur. (In larger examples, we might have additional servers, playing some part in the protocol, in addition to a watchdog that checks the correctness condition.) Figure 1 illustrates state machines for these processes.

Each process's state can be thought of as the combination of a *control state* and a vector of zero or more *parameters*, each of which is a *component identity*, either its own identity or that of another component. In more interesting examples, these parameters can be passed on to a third party. Processes synchronise on some common events, with at most two components synchronising on each event. We want to verify such systems for an arbitrary number of replicated components.

Formally, each process is represented by a parameterised state machine.

$$\begin{tikzcd}
s_0(me) \arrow[r, "pass?other.me"] & s_1(me) \arrow[d, "enter.me"] \\
s_3(me) \arrow[u, "pass.me?other"] & s_2(me) \arrow[l, "leave.me"']
\end{tikzcd}$$

$$\begin{tikzcd}
wd_0 \arrow[r, bend left, "enter?id"] & wd_1(id) \arrow[l, bend left, "leave.id"] \arrow[r, "enter?id'"] & wd_2 \arrow[loop right, "error"]
\end{tikzcd}$$

Fig. 1. Illustration of the state machines for the toy example. The diagrams are symbolic, and parameterised by the set of component identities. For example, the latter state diagram has a state $wd_1(id)$ for each identity id; there is a transition labelled $enter.id$ from wd_0 to $wd_1(id)$ for each identity id.

Definition 1. A *state machine* is a tuple (Q, Σ, δ), where: Q is a set of *states*; Σ is a set of *visible events* with $\tau \notin \Sigma$ (τ represents an internal event); and $\delta \subseteq Q \times (\Sigma \cup \{\tau\}) \times Q$ is a transition relation.

A *parameterised state machine* is a state machine where: (1) the states Q are a subset of $S \times T^*$, for some finite set S of *control states* and some potentially infinite set T of *component identities*; and (2) the events Σ are a subset of $Chan \times T^*$, for some finite set $Chan$ of *channels*.

We sometimes write a state $(s, \mathbf{x})$ as $s(\mathbf{x})$: s is a control state, and $\mathbf{x}$ records the values of its parameters (cf. Figure 1). Similarly, we write an event $(c, \mathbf{y})$ as $c.\mathbf{y}$, and write $s(\mathbf{x}) \xrightarrow{c.\mathbf{y}} s'(\mathbf{z})$ to denote $((s, \mathbf{x}), (c, \mathbf{y}), (s', \mathbf{z})) \in \delta$.

We assume that the component identities are treated polymorphically: they can be received, stored, sent, and tested for equality; but no other operations, such as arithmetic operations, can be performed on them. Processes defined in this way are naturally symmetric. Let π be a permutation on T; we write $Sym(T)$ for the set of all such permutations. We lift π to vectors from T^* by point-wise application; we then lift it to states and events by $\pi(s(\mathbf{x})) = s(\pi(\mathbf{x}))$ and $\pi(c.\mathbf{x}) = c.\pi(\mathbf{x})$. We require each state $s(\pi(\mathbf{x}))$ to be equivalent to $s(\mathbf{x})$ but with all events renamed by π: formally the states are π-bisimilar.

Definition 2. Let $M = (Q, \Sigma, \delta)$ be a state machine, and let $\pi \in Sym(T)$. We say that $\sim\, \subseteq Q \times Q$ is a π*-bisimulation* iff whenever $(q_1, q_2) \in\, \sim$ and $a \in \Sigma \cup \{\tau\}$:

- If $q_1 \xrightarrow{a} q_1'$ then $\exists\, q_2' \in Q \cdot q_2 \xrightarrow{\pi(a)} q_2' \wedge q_1' \sim q_2'$;
- If $q_2 \xrightarrow{a} q_2'$ then $\exists\, q_1' \in Q \cdot q_1 \xrightarrow{\pi^{-1}(a)} q_1' \wedge q_1' \sim q_2'$.

Definition 3. A parameterised state machine (Q, Σ, δ) is *symmetric* if for every $\pi \in Sym(T)$, $\{(s(\mathbf{x}), s(\pi(\mathbf{x}))) \mid s(\mathbf{x}) \in Q\}$ is a π-bisimulation.

This is a natural condition. In [17], we proved that under rather mild syntactic conditions, an *arbitrary* process defined using machine-readable CSP will be symmetric in this sense. The conditions are that the definition of the process contains no constant from the type T, and that it does not use certain FDR built-in functions over sets and maps, or certain compression functions.

2.1 Systems

Each system will contain a server and some number of replicated components. We assume a *single* server here, for simplicity: a system with multiple servers can be modelled by considering the parallel composition of the servers as a single server.

Each system state contains a state for the server, and a finite multiset containing the state for each component.[1] For example, one state of the toy example is $(wd_1(T_0), \{s_2(T_0), s_0(T_1), s_0(T_2)\})$, where $\{T_0, T_1, T_2\} \subseteq T$.

Definition 4. A *system* is a tuple $(Server, Cpts, Sync, Init)$ where

1. $Server = (Q_s, \Sigma_s, \delta_s)$ is a symmetric parameterised state machine representing the server;
2. $Cpts = (Q_c, \Sigma_c, \delta_c)$ is a symmetric parameterised state machine representing each replicated component;
3. $Sync \subseteq \Sigma_c$ is a set of events that require the synchronisation of two replicated components; we require $\pi(Sync) = Sync$ for each $\pi \in Sym(T)$;
4. $Init \subseteq \mathcal{SS}$ is a set of initial states, where $\mathcal{SS} = Q_s \times \mathbb{M}(Q_c)$ denotes all possible system states.

Given such a system, a *system state* is a pair $(q_s, m) \in \mathcal{SS}$, where $q_s \in Q_s$ gives the state of the server, and $m \in \mathbb{M}(Q_c)$ gives the states of the components.

A system defines a state machine $(\mathcal{SS}, \Sigma_s \cup \Sigma_c, \delta)$, where δ is defined by the following five rules (where $\rightarrow_s$ and $\rightarrow_c$ correspond to δ_s and δ_c, respectively). The rules represent, respectively: events of just the server; events of just one component; synchronisations between the server and a single component; synchronisations between two components; and synchronisations between the server and two components.

$$\frac{q_s \xrightarrow{a}_s q_s' \quad a \in (\Sigma_s - \Sigma_c) \cup \{\tau\}}{(q_s, m) \xrightarrow{a} (q_s', m)} \qquad \frac{q_c \xrightarrow{a}_c q_c' \quad a \in (\Sigma_c - Sync - \Sigma_s) \cup \{\tau\}}{(q_s, m \uplus \{q_c\}) \xrightarrow{a} (q_s, m \uplus \{q_c'\})}$$

$$\frac{q_s \xrightarrow{a}_s q_s' \quad q_c \xrightarrow{a}_c q_c' \quad a \in (\Sigma_c - Sync) \cap \Sigma_s}{(q_s, m \uplus \{q_c\}) \xrightarrow{a} (q_s', m \uplus \{q_c'\})}$$

$$\frac{q_{c,1} \xrightarrow{a}_c q_{c,1}' \quad q_{c,2} \xrightarrow{a}_c q_{c,2}' \quad a \in Sync - \Sigma_s}{(q_s, m \uplus \{q_{c,1}, q_{c,2}\}) \xrightarrow{a} (q_s, m \uplus \{q_{c,1}', q_{c,2}'\})}$$

$$\frac{q_s \xrightarrow{a}_s q_s' \quad q_{c,1} \xrightarrow{a}_c q_{c,1}' \quad q_{c,2} \xrightarrow{a}_c q_{c,2}' \quad a \in Sync \cap \Sigma_s}{(q_s, m \uplus \{q_{c,1}, q_{c,2}\}) \xrightarrow{a} (q_s', m \uplus \{q_{c,1}', q_{c,2}'\})}$$

[1] We write $\mathbb{M}$ for a finite multiset type constructor. We mostly use set notation for multisets, but write "$\uplus$" for a multiset union.

For example, in the toy example, we can take $Cpts$ and $Server$ to be the state machines illustrated in Figure 1; $Sync$ is the set of all events on channel $pass$; $Init$ is all states with the watchdog in state wd_0, a single replicated component in state s_1, and the remaining components in state s_0 (and with components having distinct identities).

Definition 5. We define the *reachable states* $\mathcal{R}$ of a system to be those system states reachable from an initial state by zero or more transitions.

Our normal correctness condition will be that the distinguished event $error$ cannot occur.

Definition 6. A system is *error-free* if there are no reachable states ss and ss' such that $ss \xrightarrow{error} ss'$.

Our normal style will be to include a *watchdog* server, that observes (some) events by other processes, and performs the event $error$ after an erroneous trace. In [18] it is shown that an arbitrary CSP traces refinement can be encoded in this way. Hence this technique can capture an arbitrary finite-state safety property.

3 Using view abstraction

In this section we describe our application of view abstraction, adapting the techniques from [1] to our synchronous message-passing setting. Fix a system $(Server, Cpts, Sync, Init)$, and let Q_s and Q_c be the states of $Server$ and $Cpts$, respectively. Let $k \in \mathbb{Z}^+$.

A *view* of size k over Q_c is a multiset $v \in \mathbb{M}(Q_c)$ of size k. A *system view* of size k is a pair (q, v) with $q \in Q_s$ and v a view of size k. We write $\mathcal{SV}_k$ for the set of all system views of size k. Note that system states and systems views have the same type: however, the latter record only part of the full system state.

Let $\mathcal{SS}_{\geq k}$ be all system states with at least k replicated components. We define the following abstraction relation, for $(q, m) \in \mathcal{SS}_{\geq k}$ and $(q, v) \in \mathcal{SV}_k$:

$$(q, v) \sqsubseteq_k (q, m) \text{ iff } v \subseteq m.$$

The system view (q, v) records the states of just k of the components of (q, m).

The abstraction function $\alpha_k : \mathcal{SS}_{\geq k} \to \mathbb{P}(\mathcal{SV}_k)$ abstracts a system state by its system views of size k:

$$\alpha_k(q, m) = \{(q, v) \in \mathcal{SV}_k \mid (q, v) \sqsubseteq_k (q, m)\}.$$

We lift α_k to sets of system states by pointwise application.

The concretization function $\gamma_k : \mathbb{P}(\mathcal{SV}_k) \to \mathcal{SS}_{\geq k}$ takes a set SV of system views, and produces those system states that are consistent with SV, i.e. such that all views of the state of size k are in SV.

$$\gamma_k(SV) = \{(q, m) \in \mathcal{SS}_{\geq k} \mid \alpha_k(q, m) \subseteq SV\}.$$

The following lemma is proved as in [1].

Lemma 7. *(α_k, γ_k) forms a Galois connection: if $A \subseteq \mathcal{SS}_{\geq k}$ and $B \subseteq \mathcal{SV}_k$, then $\alpha_k(A) \subseteq B \Leftrightarrow A \subseteq \gamma_k(B)$.*

We define an abstract transition relation. If $SV \subseteq \mathcal{SV}_k$ and $sv' \in \mathcal{SV}_k$ then define

$$SV \xrightarrow{a}_k sv' \Leftrightarrow \exists\, ss \in \gamma_k(SV)\,;\, ss' \in \mathcal{SS} \bullet ss \xrightarrow{a} ss' \wedge sv' \sqsubseteq_k ss'.$$

For example, in the running example we have the transition

$$\begin{array}{l} \{\, (wd_0, \{s_3(T_0), s_0(T_1)\}),\ (wd_0, \{s_3(T_0), s_0(T_2)\}),\ (wd_0, \{s_0(T_1), s_0(T_2)\})\,\} \\ \quad \xrightarrow{pass.T_0,T_1}_2 (wd_0, \{s_0(T_0), s_1(T_1)\}) \end{array}$$

corresponding to the concrete transition

$$(wd_0, \{s_3(T_0), s_0(T_1), s_0(T_2)\}) \xrightarrow{pass.T_0,T_1} (wd_0, \{s_0(T_0), s_1(T_1), s_0(T_2)\}).$$

We then define the abstract post-image of a set of system views $SV \subseteq \mathcal{SV}_k$ by

$$aPost_k(SV) = \{sv' \mid \exists\, a \bullet SV \xrightarrow{a}_k sv'\} = \alpha_k(post(\gamma_k(SV))),$$

where *post* gives the concrete post-image of a set $X \subseteq \mathcal{SS}$:

$$post(X) = \{(s', m') \mid \exists\, a, (s, m) \in X \bullet (s, m) \xrightarrow{a} (s', m')\}.$$

The following lemma relates abstract and concrete post-images; it is easily proved using Lemma 7.

Lemma 8. *If $SV \subseteq \mathcal{SV}_k$ and $X \subseteq \gamma_k(SV)$, then $post(X) \subseteq \gamma_k(aPost_k(SV))$.*

Let $Init_{\geq k}$ and $\mathcal{R}_{\geq k}$ be, respectively, those initial states from $Init$, and those reachable states from $\mathcal{R}$, with at least k replicated components. The following theorem shows how $\mathcal{R}_{\geq k}$ can be over-approximated by iterating the abstract post-image. We write $f^*(X)$ for $\bigcup_{i=0}^{\infty} f^i(X)$.

Theorem 9. *If $AInit \subseteq \mathcal{SV}_k$ is such that $\alpha_k(Init_{\geq k}) \subseteq AInit$ then*

$$\mathcal{R}_{\geq k} \subseteq \gamma_k(aPost_k^*(AInit)).$$

Proof: The assumption implies $Init_{\geq k} \subseteq \gamma_k(AInit)$, from Lemma 7. Then Lemma 8 implies $post^n(Init_{\geq k}) \subseteq \gamma_k(aPost^n(AInit))$ via a trivial induction. The result then follows from the fact that $\mathcal{R}_{\geq k} = post^*(Init_{\geq k})$. $\square$

Hence, if we can show that all states in $\gamma_k(aPost_k^*(AInit))$ are error-free, then we will be able to deduce that all systems with k or more components are error-free; systems with fewer than k components can be checked directly (for a fixed set of parameters, and appealing to symmetry).

In the running example, we can take $AInit$ to contain all system views of size k with the watchdog in state wd_0, zero or one components in state s_1, and the remaining components in state s_0 (and with components having distinct identities). Then, for $k \geq 2$, $\gamma_k(aPost_k^*(AInit))$ contains all system views as follows: (1) at most one component is in state s_1, s_2 or s_3, and the remainder are in s_0; and (2) if component id is in s_2 then the watchdog is in $wd_1(id)$; if every component is in s_0 then the watchdog is in either wd_0 or $wd_1(id)$ where component id is not in the view; and otherwise the watchdog is in wd_0. This approximates the invariant that a single component holds the token, and the watchdog records the component in the critical region. In particular, the event *error* is not available from any such state. The above theorem then shows that all systems of size at least two are error-free.

However, the above theorem does not immediately give an algorithm. The application of γ_k within $aPost$ can produce an infinite set, for two reasons:

- It can give system states with an arbitrary number of components;
- The parameters of type T within system states can range over a potentially infinite set.

We tackle the former problem in Section 4, by showing that it is enough to build concretizations of size at most $k+2$. We tackle the latter problem in Section 5, using symmetry.

4 Bounding the number of components

We now show that, when calculating $aPost_k$, it is enough to consider concretizations with at most two additional component states.

For $k \le l$ and $SV \subseteq \mathcal{SV}_k$, define

$$\gamma_k^l(SV) = \{(q, m_c) \in \mathcal{SS} \mid \alpha_k(q, m_c) \subseteq SV \wedge k \le \#m_c \le l\},$$

i.e., those concretizations with between k and l component states. For $k \le l$, $SV \subseteq \mathcal{SV}_k$ and $sv' \in \mathcal{SV}_k$, define the abstract transitions involving such concretizations as follows:

$$SV \xrightarrow{a}{}_k^l \, sv' \Leftrightarrow \exists\, ss \in \gamma_k^l(SV)\,;\, ss' \in \mathcal{SS} \bullet ss \xrightarrow{a} ss' \wedge sv' \sqsubseteq_k ss'.$$

Lemma 10. *Suppose $SV \subseteq \mathcal{SV}_k$, $sv' \in \mathcal{SV}_k$, $k \ge 1$, and $SV \xrightarrow{a}_k sv'$. Then $SV \xrightarrow{a}{}_k^{k+2} sv'$.*

Proof: If $SV \xrightarrow{a}_k sv' = (q_s', v')$ then for some $(q_s, m) \in \gamma_k(SV)$ and some (q_s', m') we have $(q_s, m) \xrightarrow{a} (q_s', m')$ and $sv' \sqsubseteq_k (q_s', m')$. Let $\hat{m}'$ be the smallest subset of m' that includes v' and each of the (at most two) replicated components that change state in the transition; and let $\hat{m} \subseteq m$ be the pre-transition states of the components in $\hat{m}'$. For example, suppose the transition corresponds to the fourth rule in Definition 4, so, for some m_0, $m = m_0 \uplus \{q_{c,1}, q_{c,2}\}$ and $m' = m_0 \uplus \{q_{c,1}', q_{c,2}'\}$; and suppose v' contains $q_{c,1}'$ but not $q_{c,2}'$; then $\hat{m}' = v' \uplus \{q_{c,2}'\} \subseteq m'$; and $\hat{m} \subseteq m$ is the same as $\hat{m}'$ but with $q_{c,1}$ and $q_{c,2}$ in place of $q_{c,1}'$ and $q_{c,2}'$.

In each case, it is easy to see that $(q_s, \hat{m}) \xrightarrow{a} (q_s', \hat{m}')$, via the same transition rule that produced the original transition. Also $sv' = (q_s', v') \sqsubseteq_k (q_s', \hat{m}')$. And

$k \leq \#\hat{m} = \#\hat{m}' \leq k+2$, since we have added at most two components to v'. Finally, $\hat{m} \subseteq m$, so $\alpha_k(q_s, \hat{m}) \subseteq \alpha_k(q_s, m) \subseteq SV$, so $(q_s, \hat{m}) \in \gamma_k^{k+2}(SV)$. Hence $SV \xrightarrow{a}{}_k^{k+2} sv'$. $\square$

Abdulla et al. [1] prove a similar result in their setting, although using concretizations of size at most $k+1$. We require concretizations of size $k+2$, essentially because of the possibility of a three-way synchronisation between the server and two component states (corresponding to the fifth transition rule in Definition 4). The following lemma shows that when we remove the possibility of such synchronisations, we also obtain a limit of $k+1$. However, the result is weakened to include the possibility that the system view produced was in the initial set of system views.

Lemma 11. *Suppose $Sync \cap \Sigma_s = \{\}$. Suppose further that $SV \subseteq \mathcal{SV}_k$, $sv' \in \mathcal{SV}_k$, $k \geq 1$, and $SV \xrightarrow{a}_k sv'$. Then either $sv' \in SV$ or $SV \xrightarrow{a}{}_k^{k+1} sv'$.*

Proof: The only cases in the proof of Lemma 10 where concretizations of size $k+2$ were required were transitions involving *two* replicated components —so via the fourth and fifth transition rules— where neither component state was included in sv'. The case of the fifth rule is prevented by the assumption of this lemma. In the remaining case, we have (using identifiers as in the proof of Lemma 10): $a \in Sync$, $q_s = q_s'$, $m = m_0 \uplus \{q_{c,1}, q_{c,2}\}$, $m' = m_0 \uplus \{q'_{c,1}, q'_{c,2}\}$, $v' \subseteq m_0$, $q_{c,1} \xrightarrow{a}_c q'_{c,1}$, and $q_{c,2} \xrightarrow{a}_c q'_{c,2}$. But then $sv' = (q_s, v') \sqsubseteq_k (q_s, m)$ so $sv' \in SV$. $\square$

The above lemmas show that, in order to calculate $aPost_k$ (as required for Theorem 9) it is enough to calculate either $aPost_k^{k+2}$ or (if $Sync \cap \Sigma_s = \{\}$) $aPostId_k^{k+1}$ where

$$aPost_k^l(SV) = \alpha_k(post(\gamma_k^l(SV))),$$
$$aPostId_k^l(SV) = \alpha_k(post(\gamma_k^l(SV))) \cup SV.$$

The result below follows easily from Lemmas 10 and 11.

Corollary 12. *Let $SV \subseteq \mathcal{SV}_k$ and $k \geq 1$. Then*

1. $aPost_k^*(SV) = (aPost_k^{k+2})^*(SV)$;
2. *If $Sync \cap \Sigma_s = \{\}$ then* $aPost_k^*(SV) \subseteq (aPostId_k^{k+1})^*(SV)$.

5 Using symmetry

The abstract transition relation from the previous section still produces a potentially infinite state space, because of the potentially unbounded set of component identities. In this section, we use techniques based on symmetry reduction to reduce this to a finite state space. We fix a system, as in Definition 4.

Recall (Definitions 3 and 4) that we assume that the server and each replicated component is symmetric. We show that this implies that the system as a whole is symmetric. We lift permutations to system states by point-wise application: $\pi(q, m) = (\pi(q), \{\pi(q_c) \mid q_c \in m\})$.

Lemma 13. *The state machine defined by a system is symmetric: if $(q, m) \in \mathcal{SS}$ and $\pi \in Sym(T)$, then $(q, m) \sim_\pi \pi(q, m)$.*

Proof: We show that the relation $\{((q, m), \pi(q, m)) \mid (q, m) \in \mathcal{SS}\}$ is a π-bisimulation. Suppose $(q, m) \xrightarrow{a} (q', m')$. We show that $\pi(q, m) \xrightarrow{\pi(a)} \pi(q', m')$ by a case analysis on the rule used to produce the former transition. For example, suppose the transition is produced by the third rule, so is of the form

$$(q, m_1 \uplus \{q_c\}) \xrightarrow{a} (q', m_1 \uplus \{q'_c\}),$$

such that $q \xrightarrow{a}_s q'$, $q_c \xrightarrow{a}_c q'_c$ and $a \in (\Sigma_c - Sync) \cap \Sigma_s$. Then since *Server* and *Cpts* are symmetric, and $\pi(Sync) = Sync$, we have $\pi(q) \xrightarrow{\pi(a)}_s \pi(q')$, $\pi(q_c) \xrightarrow{\pi(a)}_c \pi(q'_c)$ and $\pi(a) \in (\Sigma_c - Sync) \cap \Sigma_s$. But then

$$\pi(q, m_1 \uplus \{q_c\}) \xrightarrow{\pi(a)} \pi(q', m_1 \uplus \{q'_c\}),$$

using the same rule. The cases for other rules are similar. And conversely, we can check that each transition of $\pi(q, m)$ is matched by a transition of (q, m). □

We now show a similar result for the abstract transition relation. We lift π to system views and sets of system views by point-wise application. The following lemma shows that abstract transitions from π-related sets are related in the obvious way; it is proven using Lemma 13 and straightforward properties of permutations.

Lemma 14. *If $SV \xrightarrow{a}{}^{l}_{k} sv'$ then $\pi(SV) \xrightarrow{\pi(a)}{}^{l}_{k} \pi(sv')$.*

Our approach will be to treat symmetric system views as equivalent, requiring the exploration of only one system view in each equivalence class. We will need the following definition and lemma.

Definition 15. Let $sv_1, sv_2 \in \mathcal{SV}_k$. We write $sv_1 \approx sv_2$ if $sv_1 = \pi(sv_2)$ for some $\pi \in Sym(T)$. Note that this is an equivalence relation. We say that sv_1 and sv_2 are *equivalent* in this case.

Let $SV_1, SV_2 \subseteq \mathcal{SV}_k$. We write $SV_1 \precsim SV_2$ if

$$\forall\, sv_1 \in SV_1 \bullet \exists\, sv_2 \in SV_2 \bullet sv_1 \approx sv_2.$$

We write $SV_1 \approx SV_2$, and say that SV_1 and SV_2 are *equivalent*, if $SV_1 \precsim SV_2$ and $SV_2 \precsim SV_1$. This is again an equivalence relation.

Lemma 16. *Suppose $SV_1, SV_2 \subseteq \mathcal{SV}_k$ with $SV_1 \approx SV_2$. Then $aPost^l_k(SV_1) \approx aPost^l_k(SV_2)$.*

Proof: This follows directly from Lemma 14. □

6 The algorithm and implementation

We now present our algorithm, and prove its correctness. The algorithm takes as inputs a system, a positive integer k, and a set $AInit$ of initial system views such that $\alpha_k(Init_{\geq k}) \lesssim AInit$. If $Sync \cap \Sigma_s = \{\}$ then let $l = k+1$; otherwise let $l = k+2$. The algorithm iterates $aPost_k^l$, maintaining a set $SV \subseteq \mathcal{SV}_k$, which stores the system views encountered so far, up to equivalence.

```
$SV := AInit$
while($true$){
  if $SV \xrightarrow{error}{}_k^l$ then return $failure$
  for($sv' \in aPost_k^l(SV)$) if $\nexists sv \in SV \bullet sv \approx sv'$ then $SV := SV \cup \{sv'\}$
  if no new view was added to $SV$ then return $success$
}
```

When this algorithm is run on the toy example with $k = 2$, it encounters just five system views:

$$(wd_0, \{s_1(T_0), s_0(T_1)\}),\ (wd_0, \{s_0(T_0), s_0(T_1)\}),$$
$$(wd_1(T_0), \{s_2(T_0), s_0(T_1)\}),\ (wd_1(T_0), \{s_0(T_1), s_0(T_2)\}),\ (wd_0, \{s_3(T_0), s_0(T_2)\})$$

(or equivalent system views), the former two being the initial system views.

Lemma 17. *If the algorithm does not return $failure$ then the final value of SV is such that $\mathcal{R}_{\geq k} \lesssim \gamma_k(SV)$.*

Proof: We show that after n iterations, $SV \approx (aPostId_k^l)^n(AInit)$, by induction on n. The base case is trivial. For the inductive case, suppose, at the start of an iteration, $SV \approx (aPostId_k^l)^n(AInit)$. Each element sv' of $aPost_k^l(SV)$ is added to SV, unless SV already contains an equivalent system view. Hence the subsequent value of SV is equivalent to the value of $SV \cup aPost_k^l(SV)$ at the beginning of the iteration. But

$$\begin{aligned} SV \cup aPost_k^l(SV) &\approx (aPostId_k^l)^n(AInit) \cup aPost_k^l((aPostId_k^l)^n(AInit)) \\ &= (aPostId_k^l)^{n+1}(AInit), \end{aligned}$$

using the inductive hypothesis and Lemma 16, as required.

$\mathcal{SV}_k$ contains a finite number of equivalence classes. Hence the iteration must reach a fixed point such that SV is equivalent to $(aPostId_k^l)^*(AInit) = \bigcup_{n=0}^{\infty}(aPostId_k^l)^n(AInit)$. By Corollary 12, this contains $aPost_k^*(AInit)$. And by Theorem 9, $\mathcal{R}_{\geq k} \subseteq \gamma_k(aPost_k^*(AInit))$. □

Theorem 18. *If the algorithm returns success, then the system is error-free for systems of size at least k.*

Proof: We prove the contra-positive: suppose there is some system state $ss \in \mathcal{R}_{\geq k}$ such that $ss \xrightarrow{error}$; we show that the algorithm returns $failure$. From Lemma 17, for the fixed point of SV, we have $ss \in \gamma_k(SV)$, and so $SV \xrightarrow{error}{}_k$. Then by Lemmas 10 and 11, $SV \xrightarrow{error}{}_k^l$. Hence the algorithm returns $failure$. □

Of course, the algorithm may sometimes return $failure$ when, in fact, all systems are error-free: a *spurious counterexample*. This might just mean that it is necessary to re-run the algorithm with a larger value of k: the current value of k is not large enough to capture relevant properties of the system. Or it might be that the algorithm would fail for all values of k. This should not be surprising, since the problem is undecidable in general.

6.1 Prototype implementation

We have created a prototype implementation, in Scala, following the above algorithm[2]. Unlike the model in earlier sections, the implementation allows multiple servers: the parallel composition of these can be considered as a single server, for compatibility with the model. The current implementation supports only the conditions of Lemma 11, corresponding to $l = k + 1$; in practice, nearly all examples fit within this setting.

The implementation takes as input a value for k, and a description of the system modelled in machine-readable CSP (CSP_M): more precisely, it takes a standard CSP_M script, suitable for model checking using FDR [16], augmented with annotations to identify the processes representing the replicated components and the servers (with their initial states), their alphabets, and the type T of components' identities. CSP_M is a very expressive language, which makes it convenient for defining systems. The script must contain a concrete definition for T that is big enough, in a sense that we make clear below.

The initial state $aInit$ of the components and servers should be such that $\alpha_k(Init_{\geq k}) \subsetneq \{aInit\}$. A common case is that each initial state in $Init_{\geq k}$ contains some small number n of components in distinguished states (in the toy example, a single component in state s_1, holding the token), and all other components in some default state (in the toy example, state s_0, not holding the token), possibly with servers holding the identities of components in distinguished states. In this case, it is enough for the initial state to include the n components in distinguished states (with servers holding their identities, if appropriate), plus k components in the default state.

The program interrogates FDR to obtain state machines for the servers and components (based upon the concrete definition for T), and to check that they are symmetric in T. Using the implementation of symmetry reduction from [17], each state is represented by a control state (an integer) and a list of parameters (each an integer). From these, the program can calculate transitions from concrete system states.

The program then follows the algorithm from Section 6 quite closely. When a concretization of size l is produced, it is possible that the concretization contains more identities than were included in the concrete definition of T; in this case, the program gives an error, and the user must provide a larger type.

[2] The implementation and the scripts for the examples in the next section are available from `www.cs.ox.ac.uk/people/gavin.lowe/ViewAbstraction/index.html`.

Internally, a view (a multiset of states) is represented by a list; a system view is then represented by a list of the states of the servers (in some standard order) and this view. Hence testing whether two system views are equivalent corresponds to testing whether there is some way of permuting the view list and uniformly replacing component identities so as to make the system views equal. To make this efficient, each system view is replaced by an equivalent system view where the control states of components are in non-decreasing order, the identities are an initial segment of the natural numbers, and their first occurrences in the representation are in increasing order. The set of system views (the set SV of Section 6) is then stored as a mapping, with each system view keyed against its control states; to test whether a particular system view is equivalent to an existing one, it is enough to compare against those with the same key.

7 Analysing reference-linked data structures

We now show how our technique can be used to analyse a reference-linked data structure, such as a linked list. We illustrate our technique be verifying a lock-based concurrent queue, that uses an unbounded linked list, and that is used by two threads. We outline possible extensions to this setting in the conclusions. The queue contains data taken from the set $\{A, B, C\}$; we justify this choice below.

Each node in the linked list is modelled by a component process, and can be defined using CSP notation as follows.

$$\begin{aligned} FreeNode(me) &= initNode?t!me?d \to Node_d(me, null), \\ Node_d(me, next) &= getDatum?t!me!d \to Node_d(me, next) \\ &\quad \Box\ getNext?t!me!next \to Node_d(me, next) \\ &\quad \Box\ setNext?t!me?newNext \to Node_d(me, newNext) \\ &\quad \Box\ freeNode?t!me \to FreeNode(me). \end{aligned}$$

The state $FreeNode(me)$ represents a free node with identity me: it can be initialised by any thread t to store datum d and to have next reference to a distinguished value $null$. The state $Node_d(me, next)$ represents a node with identity me holding datum d and with next reference $next$ (we write d as a subscript, since this is not from the type of node identities, so not a parameter in the sense of the model). In this state, a thread t may: get the datum d; get the next reference $next$; set the next reference to a new value $newNext$; or free the node. Thus, nodes may be joined together to form a linked list.

In the initial state, a single node is initialised as a dummy header node in the state $Node_A(N_0, null)$, and the remaining nodes are initialised in the $FreeNode$ state.

The system contains three server processes representing part of the datatype: a lock process, that allows a thread to lock the queue; and two processes representing shared variables referencing the dummy header node, and the last node in the list, respectively, each initially holding N_0. Further, the system contains two server processes representing threads operating on the queue, enqueueing and dequeueing values (a dequeue on an empty queue returns a special value). These processes are defined as expected.

In order to verify that the system forms a queue, we adapt ideas from Wolper [30]. A process is *data independent* in a particular type D if the only operations it can perform on values of that type are to input them, store them, and output them. This means that for each trace tr of the process, uniformly replacing values from D within tr will give another trace of the process.

Lemma 19. *Suppose a process is data independent in a type D. Suppose further that whenever a sequence of data values from the language A^*BC^* is enqueued, then a sequence from $A^*BC^* + A^*$ is dequeued, and no dequeue operation finds the queue empty between the enqueue and dequeue of B. Then it is a queue.*

Proof: (sketch). Consider a behaviour of such a process that violates the property of being a queue, by either losing, duplicating or reordering a particular piece of data. Then, by data independence, a similar behaviour would occur on an input from A^*BC^*, losing, duplicating or reordering B. But this would produce an output not from this language, or a dequeue would find the queue empty between the enqueue and dequeue of B. $\square$

We add two servers so as to exploit this idea:

- A regulator process, that synchronises with the threads, to force them to enqueue a sequence from A^*BC^*;
- A watchdog process, that observes the values dequeued, and performs $error$ if the sequence is not from $A^*BC^* + A^*$, or if a dequeue finds the queue empty between the enqueue and dequeue of B.

The prototype implementation can be used to explore this system: the test succeeds in the case $k = 2$, and completes in about 12 seconds. Hence, by Theorem 18, all systems with at least two nodes implement a queue (for two threads). It is necessary to include at least nine values in the type of node identities: when considering transitions from states of size $k + 1 = 3$ (cf. Lemma 11), system states are encountered with three nodes, each holding their own identity and one other; the $Header$ and $Tail$ processes can each hold one other identity; and the thread holding the lock can hold one of these and one other identity.

We have used similar ideas to analyse a lock-based stack that uses a linked list. The modelling is very similar to as for the queue. For verification, we ensure that the values pushed onto the stack form a sequence from A^*BC^*; we then check (using a watchdog) that (1) before B is pushed, only A can be popped; (2) after B is pushed, the sequence of values popped is from $C^*B(A + C)^*$; and (3) a pop does not find the stack empty between the B being pushed and popped. An argument similar to Lemma 19 justifies the correctness of this test. The analysis, with $k = 2$, takes about 10 seconds in this case.

8 Conclusions

In this paper we have tackled a particular instance of the parameterised model checking problem, where replicated component processes have identities that may be passed between processes. We have adapted the technique of view abstraction, which records, for each system state, the states of just some small number k of replicated components. We have used techniques from symmetry reduction, to bound the number of identities of components that are stored. We have provided an implementation based on systems defined in CSP (although the underlying ideas are not CSP-specific). We have shown that the framework allows us to analyse unbounded reference-linked datatypes.

Roughly speaking, our technique, with a particular value of k, succeeds for systems whose invariant can be described in terms of the states of servers and at most k replicated components. For example, with the queue of Section 7, when a sequence from A^*BC^* is enqueued, each pair of adjacent nodes in the linked list hold data values (A, A), (A, B), (B, C) or (C, C), which implies that the sequence held is from $A^* + A^*BC^* + C^*$: this invariant talks about the states of just two components, so taking $k = 2$ succeeds.

Wolper [30] uses a technique similar to ours for characterising queues, but based on enqueueing a sequence from $A^*BA^*CA^*$. Curiously, our approach will not work with such a sequence, and gives a spurious error. This is because the corresponding invariant cannot be described in terms of the states of a bounded number of components, because a node holding A can be followed by a node holding any datum. This suggests that when trying to characterise a particular datatype based on chosen input sequences, those sequences should not contain the same data value in two different "chunks".

In this paper we have assumed a *single* family of replicated components. We intend to extend this to multiple such families. For example, in Section 7, we could have considered a family of processes representing the threads that interact with the queue, to allow us to verify that the datatype is correct when used by an arbitrary number of threads.

Our main motiviating domain for this work is the study of concurrent datatypes, particularly lock-free datatypes. In [21] we used CSP and FDR to analyse a lock-free queue based on a linked list [24] for a *fixed* number of nodes and threads. We would like to use the techniques from this paper to consider an *arbitrary* number of nodes and threads. The main challenge here is capturing the correctness condition of linearizability: we believe this will be straightforward when explicit linearization points are given, but harder otherwise.

We have assumed a fully connected topology, where each replicated component can communicate with each other. We intend to also consider more restrictive topologies, such as a ring, following [1, Section 3.4].

In this paper we have considered only safety properties, corresponding to traces of the system. We would like to be able to consider also liveness properties, such as deadlock-freedom. One can adapt the algorithm from Section 6 to test whether any concretization of the set of system views deadlocks; one can then prove a variant of Theorem 18 that shows that if no such deadlock is found, then no system of size at least k deadlocks. However, this does not work in practice, since the abstraction introduces too many spurious deadlocks that do not occur in the unabstracted system. We intend to investigate whether other abstractions work better for this purpose.

Acknowledgements I would like to thank Tom Gibson-Robinson for useful discussions concerning this work, and for extending the FDR4 API to support various functions necessary for the implementation from Section 6.1.

References

1. P. Abdulla, F. Haziza, and L. Holík. Parameterized verification through view abstraction. *International Journal on Software Tools for Technology Transfer*, 18:495–516, 2016.
2. P. Abdulla, B. Jonsson, M. Nilsson, and J. d'Orso. Regular model checking made simple and efficient. In *Proceedings of CONCUR'02, 13th International Conference on Concurrency Theory*, volume 2421 of *LNCS*, pages 116–130, 2002.
3. P. Abdulla, B. Jonsson, M. Nilsson, and M. Saksena. A survey of regular model checking. In *Proceedings of Concur*, volume 3170 of *LNCS*, pages 35–48, 2004.
4. P. Abdullah, F. Haziza, L. Holík, B. Jonsson, and A. Rezine. An integrated specification and verification technique for highly concurrent data structures. *International Journal on Software Tools for Technology Transfer*, 19:549–563, 2017.
5. K. R. Apt and D. C. Kozen. Limits for automatic verification of finite-state concurrent systems. *Information Processing Letters*, 22(6):307–309, 1986.
6. A. Bouajjani, P. Habermehl, and T. Vojnar. Abstract regular model checking. In *Proceedings of International Conference on Computer Aided Verification*, volume 3114 of *LNCS*, pages 372–386, 2004.
7. A. Bouajjani, B. Jonsson, M. Nilsson, and T. Touili. Regular model checking. In *Proceedings of the 12th International Conference on Computer Aided Verification*, volume 1855 of *LNCS*, pages 403–418, 2000.
8. D. Bošnački, D. Dams, and L. Holenderski. Symmetric Spin. *International Journal on Software Tools for Technology Transfer*, 4:92–106, 2002.
9. E. M. Clarke, E. A. Emerson, S. Jha, and A. P. Sistla. Symmetry reductions in model checking. In *Proceedings of the 10th International Conference on Computer-Aided Verification (CAV '98)*, pages 147–158, 1998.
10. E. M. Clarke, R. Enders, T. Filkorn, and S. Jha. Exploiting symmetry in temporal logic model checking. *Formal Methods in System Design*, 9:77–104, 1996.
11. E. M. Clarke and O. Grumberg. Avoiding the state explosion problem in temporal logic model checking. In *Proceedings of the 6th Annual Association for Computing Machinery Symposium on Principles of Distributed Computing*, pages 294–303, 1987.
12. D. Dams, Y. Lakhnech, and M. Steffen. Iterating transducers. *The Journal of Logic and Algebraic Programming*, 52–53:109–127, 2002.
13. E. A. Emerson and K. S. Namjoshi. Reasoning about rings. In *Proceedings of the Symposium on Principles of Programming Languages (POPL '95)*, 1995.
14. E. A. Emerson and A. P. Sistla. Symmetry and model checking. *Formal Methods in System Design*, 9:105–131, 1996.
15. S. M. German and A. P. Sistla. Reasoning about systems with many processes. *Journal of the ACM*, 39(3):675–735, 1992.

16. T. Gibson-Robinson, P. Armstrong, A. Boulgakov, and A. W. Roscoe. FDR3: a parallel refinement checker for CSP. *International Journal on Software Tools for Technology Transfer*, 2015.
17. T. Gibson-Robinson and G. Lowe. Symmetry reduction in CSP model checking. Submitted for publication. Extended version at `http://www.cs.ox.ac.uk/people/gavin.lowe/SymmetryReduction/`, 2017.
18. M. Goldsmith, N. Moffat, B. Roscoe, T. Whitworth, and I. Zakiuddin. Watchdog transformations for property-oriented model checking. In *Proceedings of Formal Methods Europe (FME 2003)*, volume 2805 of *LNCS*, pages 600–616, 2003.
19. C. N. Ip and D. L. Dill. Better verification through symmetry. *Formal Methods in System Design*, 9:41–75, 1996.
20. Y. Kesten, O. Maler, M. Marcus, A. Pnueli, and E. Shahar. Symbolic model checking with rich assertional languages. *Theoretical Computer Science*, pages 93–112, 2001.
21. G. Lowe. Analysing lock-free linearizable datatypes using CSP. In *Concurrency, Security and Puzzles: Essays Dedicated to Andrew William Roscoe on the Occasion of His 60th Birthday*, volume 10160 of *Lecture Notes in Computer Science*, pages 162–184. Springer, 2017.
22. B. Lubachevsky. An approach to automating the verification of compact parallel coordination programs. *Acta Informatica*, 21(2):125–169, 1984.
23. T. Mazur and G. Lowe. CSP-based counter abstraction for systems with node identifiers. *Science of Computer Programming*, 81:3–52, 2014.
24. M. Michael and M. Scott. Simple, fast, and practical non-blocking and blocking concurrent queue algorithms. In *Proc. of the Fifteenth Annual ACM Symposium on Principles of Distributed Computing*, pages 267–275, 1996.
25. A. Pnueli, J. Xu, and L. D. Zuck. Liveness with $(0, 1, \infty)$-counter abstraction. In *CAV'02: Proceedings of the 14th International Conference on Computer Aided Verification*, pages 107–122, 2002.
26. A. W. Roscoe. *Understanding Concurrent Systems*. Springer, 2010.
27. A. W. Roscoe and S. Creese. Data independent induction over structured networks. In *Proceedings of PDPTA2000*, 2000.
28. A. P. Sistla, V. Gyuris, and E. A. Emerson. SMC: A symmetry-based model checker for verification of safety and linveness properties. *ACM Transactions on Software Engineering and Methodology*, 9(2):133–166, 2000.
29. T. Touili. Regular model checking using widening techniques. In *Electronic Notes in Theoretical Computer Science, 50(4), Proceedings of VEPAS'01*, 2001.
30. P. Wolper. Expressing interesting properties of programs in propositional temporal logic. In *Thirteenth Annual ACM Symposium on Principles of Programming Languages*, pages 184–193, 1986.
31. P. Wolper and B. Boigelot. Verifying systems with infinite but regular state spaces. In *Proceedings of 10th International Conference on Computer Aided Verification*, volume 1427 of *LNCS*, 1998.
32. P. Wolper and V. Lovinfosse. Verifying properties of large sets of processes with network invariants. In *Automatic Verification Methods for Finite State Systems*, volume 407 of *LNCS*, pages 68–80, 1989.