The automatic detection of token structures and invariants using SAT checking

Pedro Antonino, Thomas Gibson-Robinson, and A.W. Roscoe

Department of Computer Science, University of Oxford, Oxford, UK
{pedro.antonino,thomas.gibson-robinson,bill.roscoe}@cs.ox.ac.uk

Abstract. Many distributed systems rely on token structures for their correct operation. Often, these structures make sure that a fixed number of tokens exists at all times, or perhaps that tokens cannot be completely eliminated, to prevent systems from reaching undesired states. In this paper we show how a SAT checker can be used to automatically detect token and similar invariants in distributed systems, and how these invariants can improve the precision of a deadlock-checking framework that is based on local analysis. We demonstrate by a series of practical experiments that this new framework is as efficient as similar incomplete techniques for deadlock-freedom analysis, while handling a different class of systems.

1 Introduction

Many concurrent and distributed systems rely on some token mechanism to avoid reaching undesired states. For these systems, understanding/recognising these token structures often leads to system invariants (i.e. system abstractions) that are sufficiently strong to prove safety properties of the considered system. For instance, token invariants are frequently used to show mutual exclusion properties and deadlock freedom. In this work, motivated by deadlock-freedom analysis, we propose two techniques that can recognise token structures using SAT checking. The first technique detects token structures where the number of tokens is conserved at all times, whereas the second one ensures that at least one token exists in the system at all times.

To demonstrate how these structures can be used in the analysis of safety properties, we combine our detection techniques with the local-analysis framework for deadlock checking presented in [3] to create a more precise, albeit still incomplete, deadlock-checking framework. Incomplete frameworks can be far more scalable than complete ones at the cost of being unable to prove that some deadlock-free systems are deadlock free. The new token framework handles a different class of system than current incomplete techniques for deadlock-freedom analysis. We implement this new framework and our detection techniques in a new mode of the DeadlOx tool [4], called *DeadlOx-VT* (for Virtual Tokens). We reinforce that the core of our framework should be easily adaptable for the verification of other safety properties using other formalisms.

Outline. Section 2 briefly introduces CSP's operational semantics, which is the formalism upon which our strategy is based. However, this paper can be understood purely in terms of communicating LTSs, and knowledge of CSP is not a prerequisite. Section 3 presents some related invariant generation and incomplete deadlock-freedom-checking techniques. In Section 4, we introduce our techniques for automatically detecting token structures. Section 5 presents our new framework for imprecise deadlock-freedom checking. Section 6 presents an experiment conducted to assess the accuracy and efficiency of DeadlOx-VT. Finally, in Section 7, we present our concluding remarks.

2 Background

The CSP notation [12, 18] models concurrent systems as processes that exchange messages. Here we describe some structures used by the refinement checker FDR3 [10] in implementing CSP's operational semantics. As this paper does not depend on the details of CSP, we do not describe the details of the language or its semantics. These can be found in [18].

FDR3 interprets CSP terms as a *labelled transition system* (LTS).

Definition 1. *A labelled transition system is a 4-tuple* $(S, \Sigma, \Delta, \hat{s})$ *where* S *is a set of states,* Σ *is the alphabet,* $\Delta \subseteq S \times \Sigma \times S$ *is a transition relation, and* $\hat{s} \in S$ *is the starting state.*

FDR3 represents concurrent systems as *supercombinator machines*. A supercombinator machine consists of a set of component LTSs along with a set of rules that describe how components transitions should be combined. We restrict FDR3's usual definition to systems with pairwise communication, as per [14, 4].

Definition 2. *A triple-disjoint supercombinator machine is a pair* $(\mathcal{L}, \mathcal{R})$ *where:*

- $\mathcal{L} = \langle L_1, \ldots, L_n \rangle$ *is a sequence of component LTSs;*
- $\mathcal{R}$ *is a set of rules of the form* (e, a) *where:*
  - $e \in (\Sigma^-)^n$ *specifies the event that each component must perform, where* $-$ *indicates that the component performs no event.* e *must also be triple-disjoint, that is, at most two components must be involved in a rule.*
    - $\mathit{triple\_disjoint}(e) \mathrel{\widehat{=}} \forall i, j, k : \{1 \ldots n\} \mid i \neq j \wedge j \neq k \wedge i \neq k \bullet e_i = - \vee e_j = - \vee e_k = -$
  - $a \in \Sigma$ *is the event the supercombinator performs.*

We say that two components interact/communicate in a supercombinator machine, if a rule in this system requires the participation of these two components. Given a supercombinator machine, a corresponding LTS can be constructed.

Definition 3. *Let* $\mathcal{S} = (\langle L_1, \ldots, L_n \rangle, \mathcal{R})$ *be a supercombinator machine where* $L_i = (S_i, \Sigma_i, \Delta_i, \hat{s}_i)$*. The LTS induced by* $\mathcal{S}$ *is the tuple* $(S, \Sigma, \Delta, \hat{s})$ *such that:*

- $S = S_1 \times \ldots \times S_n$*;*

– $\Sigma = \bigcup_{i=1}^{n} \Sigma_i$;
– $\Delta = \{((s_1, \ldots, s_n), a, (s'_1, \ldots, s'_n)) \mid \exists((e_1, \ldots, e_n), a) : \mathcal{R} \bullet \forall i : \{1 \ldots n\} \bullet$
$(e_i = - \wedge s_i = s'_i) \vee (e_i \neq - \wedge (s_i, e_i, s'_i) \in \Delta_i)\}$;
– $\hat{s} = (\hat{s}_1, \ldots, \hat{s}_n)$.

We write $s \xrightarrow{e} s'$ if $(s, e, s') \in \Delta$. There is a path from s to s' with the sequence of events $\langle e_1, \ldots, e_n \rangle \in \Sigma^*$, represented by $s \xrightarrow{\langle e_1, \ldots, e_n \rangle} s'$, if there exist $s_0, \ldots, s_n$ such that $s_0 \xrightarrow{r_1} s_1 \ldots s_{n-1} \xrightarrow{r_n} s_n$, $s_0 = s$ and $s_n = s'$.

From now on, we use *system state* (*component state*) to designate a state in the system's (component's) LTS. Also, for the sake of decidability, we only analyse supercombinator machines with a finite number of components, which are themselves represented by finite LTSs with finite alphabets.

Definition 4. *A LTS* $(S, \Sigma, \Delta, \hat{s})$ *deadlocks in a state* s *iff* $deadlock(s)$ *holds, where* $deadlock(s) \mathrel{\widehat{=}} reachable(s) \wedge blocked(s)$, $reachable(s) \mathrel{\widehat{=}} \exists t : \Sigma^* \bullet \hat{s} \xrightarrow{t} s$, *and* $blocked(s) \mathrel{\widehat{=}} \neg \exists s' : S ; e : \Sigma \bullet s \xrightarrow{e} s'$.

3 Related Work

System invariants are meant to capture compact abstractions of a system's behaviour. For concurrent and distributed systems, invariants are often calculated by combining component invariants using rules that carefully analyse how components interact [13, 5, 8, 4]. Component invariants can be automatically generated using static analysis [4] or by custom-made generation rules [8]. These automatic invariant-generation techniques tend to be either too imprecise to capture token structures in general [4], or too precise so that it captures not only token structures but a much more complex abstraction of the system [8]. Token invariants are commonly used to prove mutual-exclusion properties and deadlock-freedom for Petri nets [1, 16]. However, many systems are more naturally described by formalisms where token structures are not obviously recognisable. We are not aware of any previous use of SAT checkers to calculate token-like invariants.

In the context of deadlock-analysis, we proposed *Pair* [3], a technique that uses local analysis to check deadlock-freedom. It characterises a deadlock by analysing how pairs of components interact using the following projection:

Definition 5. *Let* $\mathcal{S} = (\langle L_1, \ldots, L_n \rangle, \mathcal{R})$ *be a supercombinator machine. The* pairwise projection $\mathcal{S}_{i,j}$ *of the machine* $\mathcal{S}$ *on components* i *and* j *is given by:*

$$\mathcal{S}_{i,j} = (\langle L_i, L_j \rangle, \{((e_i, e_j), a) \mid \exists((e_1, \ldots, e_n), a) \in \mathcal{R} \bullet (e_i \neq - \vee e_j \neq -)\})$$

Pair characterises a deadlock as a state of the system that is fully consistent with local reachability and blocking information. We call it a *Pair candidate*.

Definition 6. *Let* $\mathcal{S} = (\langle L_1, \ldots, L_n \rangle, \mathcal{R})$ *be a supercombinator machine, and* $(S, \Sigma, \Delta, \hat{s})$ *its induced LTS. A state* $s = (s_1, \ldots, s_n) \in S$ *is a* Pair candidate *iff* $pair\_candidate(s)$ *holds, where:*

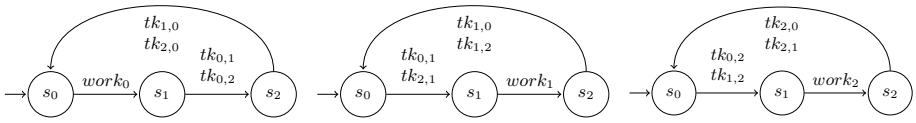

Fig. 1. LTSs of components L_0, L_1, and L_2, respectively.

- $pair\_candidate(s) \mathrel{\widehat{=}} pairwise\_reachable(s) \wedge blocked(s)$
- $pairwise\_reachable(s) \mathrel{\widehat{=}} \forall\, i, j \in \{1 \ldots n\} \mid i \neq j \bullet reachable_{i,j}((s_i, s_j))$

$reachable_{i,j}$ is the reachable predicate for the pairwise projection $\mathcal{S}_{i,j}$.

The analysis of pairs of components cannot precisely characterise reachability; Pair approximates reachability with $pairwise\_reachable(s)$. This limitation makes this technique unable to show unreachability if that is due to some global property of the system's behaviour.

To cope with this inability, some incomplete frameworks combine the use of local analysis with some system invariants [4, 15]. However, these techniques rely on a degree of predicability in how individual components interact. So, they often work well on token *rings* where tokens take a predictable route round the network, but they do not seem to do so on more complex uses of tokens. The following two deadlock-free systems employ a token mechanism where components can dynamically choose which other component to pass a token to; this unpredictability make these techniques unable to prove them deadlock free.

Running example 1. Let $\mathcal{S} = (\langle L_0, L_1, L_2 \rangle, \mathcal{R})$ be the supercombinator machine with L_0, L_1 and L_2 defined in Figure 1 and $\mathcal{R}$ the set of rules that require components to synchronise on shared events; e.g. for event $tk_{0,1}$, we have rule $((tk_{0,1}, tk_{0,1}, -), tk_{0,1})$. An arrow with two labels represents two transitions with the same source and target states but with different labels. $\mathcal{S}$ implements a token network where process L_0 has the token initially and event $tk_{i,j}$ represents the passage of a token from L_i to L_j. Both Pair and the techniques in [4] are unable to show (s_1, s_2, s_2) unreachable, so they consider it a deadlock candidate. □

Running example 2. Let $\mathcal{S} = (\langle L_0, L_1, L_2 \rangle, \mathcal{R})$ be the supercombinator machine with L_0, L_1 and L_2 defined in Figure 2 and $\mathcal{R}$ the set of rules that requires components to synchronise on shared events except for τ that can be performed independently. Component i can receive a message (i.e. a token) either from component j, via event $tk_{j,i}$, or from its user, via event in_i. If it holds a message, it can pass the message to component j, via event $tk_{i,j}$, or output the message to its user, via out_i. The τ transitions represent an internal (non-deterministic) decision of the component. Neither Pair nor the techniques in [4] can show that the state (s_6, s_6, s_6) is unreachable, so they flag it as a potential deadlock. □

4 Detecting Token Structures and Invariants using SAT

Many concurrent systems use some sort of token mechanism to guide interactions between components and avoid undesired behaviours. In this section, we present

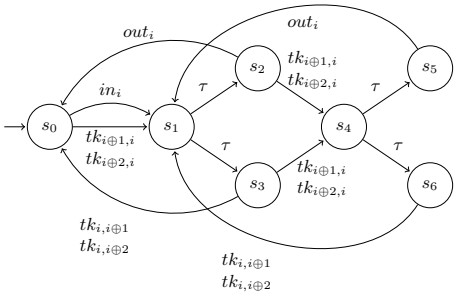

Fig. 2. LTS of component L_i where $\oplus$ represents addition modulo 3.

two techniques that interpret concurrent systems as token networks, trying to understand how *virtual* tokens might flow in these systems. We use "virtual" as tokens are not part of the system itself but rather an element of the abstract token mechanism it employs. Each technique assumes a particular policy that controls how tokens can flow. So, our techniques try to mark in which component states a component holds a token; this marking represents a token flow. This marking is later used to create reachability invariants (i.e. predicates over system states that over-approximate reachability) for the system under analysis.

4.1 Conservative Technique

Each technique proposes a SAT formula $\mathcal{F}$ with a boolean variable $t_{i,s}$ for each state s of each component i such that the values for these variables in a satisfying assignment creates a marking of the component states. The boolean value assigned to $t_{i,s}$ represents whether the component i is holding a virtual token at state s or not. $\mathcal{F}$ is a conjunction of three sub-formulas: $Policy$, $NotAlwaysHoldingToken$ and $Participation$.

$Policy$ enforces a token-flow policy; it dictates how tokens are manipulated when components (inter)act (i.e. a system transition takes place). As the system being analysed is triple disjoint, either a component acts on its own (i.e. an individual transition takes place) or a pair of components agrees on a rule and interact (i.e. a pairwise transition takes place). So, this sub-formula relies on constraint $enc_i(s, s')$ to dictate how tokens are to be manipulated by individual transitions, whereas $enc_{i,j}(s, s')$ is its counterpart for pairwise transitions.

The first technique we propose, which we refer to as the *conservative* technique, implements a token-conservation policy. For an individual transition (s, s') of component i, $enc_i(s, s')$ is as follows.

$$enc_i(s, s') \mathrel{\widehat{=}} t_{i,s} \leftrightarrow t_{i,s'} \tag{1}$$

For a pairwise transition $(s, s') \mathrel{\widehat{=}} ((s_0, s_1), (s'_0, s'_1))$ involving components i and j, $enc_{i,j}(s, s')$ is as follows. It allows exchanges of tokens between i and j.

It relies on the auxiliary variables max_{src}, min_{src}, max_{tgt}, and min_{tgt} to count the number of tokens in the source s and target s' states, respectively.

$$\begin{aligned} enc_{i,j}(s,s') \mathrel{\widehat{=}} & \; max_{src} \leftrightarrow (t_{i,s_0} \vee t_{j,s_1}) \wedge max_{tgt} \leftrightarrow (t_{i,s'_0} \vee t_{j,s'_1}) \\ & \wedge min_{src} \leftrightarrow (t_{i,s_0} \wedge t_{j,s_1}) \wedge min_{tgt} \leftrightarrow (t_{i,s'_0} \wedge t_{j,s'_1}) \\ & \wedge max_{src} \leftrightarrow max_{tgt} \wedge min_{src} \leftrightarrow min_{tgt} \end{aligned} \tag{2}$$

Policy ensures a token-policy by making sure that for all system transitions either enc_i or $enc_{i,j}$ is enforced, according to whether the transition is individual or pairwise, respectively. Thanks to triple-disjointness, the transitions of system $\mathcal{S}$ can be efficiently over-approximated by the examination of components, or rather component projections $\mathcal{S}_i$, and pairs of interacting components, or rather pairwise projections $\mathcal{S}_{i,j}$ as per Definition 5.

Definition 7. *Let $\mathcal{S} = (\langle L_1, \ldots, L_n \rangle, \mathcal{R})$ be a supercombinator machine. The* component projection *$\mathcal{S}_i$ of the machine $\mathcal{S}$ on components i is given by:*

$$\mathcal{S}_i = (\langle L_i \rangle, \{((e_i), a) \mid \exists ((e_1, \ldots, e_n), a) \in \mathcal{R} \bullet e_i \neq -\})$$

For a component projection $\mathcal{S}_i$, transitions of its induced LTS that are derived from *pure-individual* rules (i.e. rules that come from individual rules in $\mathcal{S}$) represent possible system transitions, whereas transitions derived from *truncated* rules (i.e. rules that come from pairwise rules of $\mathcal{S}$ that involve i and another component of the system) do not. For pairwise projections $\mathcal{S}_{i,j}$, only transitions derived from pairwise rules in $\mathcal{S}_{i,j}$ represent possible system transitions.

Definition 8. *Let $\mathcal{S} = (\langle L_1, \ldots, L_n \rangle, \mathcal{R})$ be a supercombinator machine, Δ_i the transition relation of the LTS induced by component projection $\mathcal{S}_i$, $\Delta_{i,j}$ the transition relation of the LTS induced by pairwise projection $\mathcal{S}_{i,j}$, and Sync the set of pairs of components interacting, i.e. participating together in a rule, in $\mathcal{S}$.*

$$Policy \mathrel{\widehat{=}} \Big(\bigwedge_{\substack{i \in \{1 \ldots n\} \\ \wedge (s,a,s') \in \Delta_i \\ \wedge ind_i(s,s')}} enc_i(s,s') \Big) \wedge \Big(\bigwedge_{\substack{(i,j) \in Sync \\ \wedge (s,a,s') \in \Delta_{i,j} \\ \wedge pair_{i,j}(s,s')}} enc_{i,j}(s,s') \Big)$$

where $ind_i(s,s')$ holds iff *(s,s') is a transition derived from a pure-individual rule of $\mathcal{S}_i$ involving component i, and $pair_{i,j}(s,s')$ holds* iff *(s,s') is a transition derived from an pairwise rule of $\mathcal{S}_{i,j}$.*

Lemma 1. *Let $\mathcal{S} = (\langle L_1, \ldots, L_n \rangle, \mathcal{R})$ be a supercombinator machine, Δ the transition relation of its induced LTS, $\Delta_{i,j}$ the transition relation of the LTS induced by component projection $\mathcal{S}_{i,j}$, and s_i the i-th element in s.*

For r a pairwise rule in which components i and j participate,
if $(s,a,s') \in \Delta$ is derived from r, then $((s_i,s_j),a,(s'_i,s'_j)) \in \Delta_{i,j}$

Proof. Follows from the definition of a pairwise projection and its LTS. □

Lemma 2. *Let $\mathcal{S} = (\langle L_1, \ldots, L_n \rangle, \mathcal{R})$ be a supercombinator machine, Δ the transition relation of its induced LTS, Δ_i the transition relation of the LTS induced by component projection $\mathcal{S}_i$, and s_i the i-th element in s.*

For r a individual rule in which component i participates,
if $(s, a, s') \in \Delta$ is derived from r, then $((s_i), a, (s'_i)) \in \Delta_i$

Proof. Follows from the definition of a component projection and its LTS. □

The sub-formulas $NotAlwaysHoldingToken$ and $Participation$ forbid some trivial markings (i.e. in which tokens do not get exchanged between components) from being valid assignments for our formula. The $NotAlwaysHoldingToken$ sub-formula forbids assignments where some component always holds a token, though we do permit components that never hold a token. $Participation$ requires the system to hold at least one token initially. To implement $Participation$, we create the participation variables p_i. In a satisfying assignment, the variable p_i states whether component i participates on the token-flow represented by this assignment. These variables play an important role as we present later.

Definition 9. *Let $\mathcal{S} = (\langle L_1, \ldots, L_n \rangle, \mathcal{R})$ be a supercombinator machine where S_i and $\hat{s}_i$ gives the set of states and the starting state of component L_i, respectively.*

$$NotAlwaysHoldingToken \mathrel{\widehat{=}} \bigwedge_{i:\{1\ldots n\}} (\bigvee_{s:S_i} \neg t_{i,s})$$

$$Participation \mathrel{\widehat{=}} \bigwedge_{i:\{1\ldots n\}} (p_i \leftrightarrow (\bigvee_{s:S_i} t_{i,s})) \wedge \bigvee_{i:\{1\ldots n\}} t_{i,\hat{s}_i}$$

For the conservative technique, we end up with the following formula:

Definition 10. *For the supercombinator machine $\mathcal{S}$,*

$$\mathcal{F} \mathrel{\widehat{=}} Policy \wedge NotAlwaysHoldingToken \wedge Participation$$

where $Policy$ uses enc_i and $enc_{i,j}$ as defined in (1) *and* (2)*, respectively.*

This technique uses function FindMarkings in Algorithm 1 to systematically find markings for different parts of our systems. For this algorithm, we use the function Solve to solve SAT formulas. It returns whether the formula is satisfiable and updates the global field $\mathcal{A}$ with a satisfying assignment. When Solve is called for an unsatisfiable formula, $\mathcal{A}$ is not updated. We use $\mathcal{A}(var)$ to denote the value assigned to variable var on the satisfying assignment $\mathcal{A}$.

The call to Solve in FindMarkings tries to find a marking for some subsystem (i.e. a subset of components) of the system $\mathcal{S}$. Note that the $Participation$ clause only requires some subsystem of $\mathcal{S}$ to participate in a token-flow. If a marking is found, it is minimised by Minimise. The minimal marking is, then, recorded by ExtractMarking. We modify our formula at the end of each iteration to ensure that in the next iteration we look for a marking for a different subsystem; this also guarantees that our function terminates.

Minimise iteratively minimises the subsystem currently marked (i.e. the components that participate in the token-flow associated with the current satisfying assignment in $\mathcal{A}$), making sure a component in this subsystem holds a token initially, until a minimal subsystem is found. It begins with the subsystem marked by FindMarkings, and at each iteration, it tries to mark a strictly smaller

Algorithm 1 Algorithm to find conservative token-structures

1: **function** FindMarkings($\mathcal{S}$)
2: $\quad$ *partition* := $\emptyset$; *marking* := $\emptyset$
3: $\quad$ Construct $\mathcal{F}$ for $\mathcal{S}$
4: $\quad$ **while** Solve($\mathcal{F}$) **do**
5: $\quad\quad$ Minimise($\mathcal{F}$)
6: $\quad\quad$ ExtractMarking($\mathcal{A}$)
7: $\quad\quad$ $\mathcal{F} := \mathcal{F} \wedge (\bigwedge_{i:\{1\ldots n\}\wedge\mathcal{A}(p_i)} \neg p_i)$
8: $\quad$ **end while**
9: **end function**

10: **function** Minimise($\mathcal{F}$)
11: $\quad$ **repeat**
12: $\quad\quad$ $\mathcal{F} := \mathcal{F} \wedge (\bigvee_{\substack{i\in\{1\ldots n\}\\ \wedge\mathcal{A}(p_i)}} \neg p_i) \wedge (\bigvee_{\substack{i\in\{1\ldots n\}\\ \wedge\mathcal{A}(p_i)}} t_{i,\hat{s}_i}) \wedge (\bigwedge_{\substack{i\in\{1\ldots n\}\\ \wedge\neg\mathcal{A}(p_i)}} \neg p_i)$
13: $\quad$ **until not** Solve($\mathcal{F}$)
14: **end function**

15: **function** ExtractMarking($\mathcal{A}$)
16: $\quad$ partitions := partitions $\cup$ $\{\{i \mid i \in \{1\ldots n\} \wedge \mathcal{A}(p_i)\}\}$
17: $\quad$ marking := marking $\cup$ $\{(i, s, \mathcal{A}(t_{i,s})) \mid i \in \{1\ldots n\} \wedge s \in S_i \wedge \mathcal{A}(p_i)\}$
18: **end function**

subsystem. Finally, ExtractMarking records in the global fields *partitions* and *marking* the subsystem marked and the marking itself.

The proposed minimisation attempts to more finely capture the behaviour of systems. Small(er) subsystems imply that we know more precisely where tokens are confined, and so, we have a better understanding on how tokens can move around. For instance, we can better identify illegal behaviours such as a token that has moved between two confined subsystems.

We use the information recorded in *partitions* and *marking* to create reachability invariants. As we enforce the preservation of the number of tokens for any system transition, all reachable states must have the same number of tokens. So, we can calculate the number of tokens at the initial state and use it to enforce this *sum invariant*; we systematically enforce it for each subsystem in *partitions*.

Definition 11. *Let $\mathcal{S} = (\langle L_1, \ldots, L_n\rangle, \mathcal{R})$ be a supercombinator machine where $\hat{s}_i$ is the starting state for L_i, partitions and marking the sets recorded after the execution of* FindMarkings($\mathcal{S}$) *in Algorithm 1, and marking(i, s) yields* 1 *if the state s of component i is assigned to true, and* 0 *otherwise. The reachability invariant $reach_C(s)$ is as follows:*

$$reach_C(s) \mathrel{\widehat{=}} \forall\, sub \in partitions \bullet N(sub) = Tks(sub, s)$$

where $N(sub) \mathrel{\widehat{=}} \sum_{i\in sub} marking(i, \hat{s}_i)$, *and* $Tks(sub, s) \mathrel{\widehat{=}} \sum_{i\in sub} marking(i, s_i)$.

Lemma 3. $reachable(s) \Rightarrow reach_C(s)$

Proof. Let *sub* be a subsystem in *partitions*. For $\hat{s}$ the starting state of the LTS induced by system $\mathcal{S}$, we have trivially that $N(sub) = Tks(sub, \hat{s})$. Each transition preserves this invariant, that is, if $(s, a, s') \in \Delta$ then $Tks(sub, s) = Tks(sub, s')$. This is guaranteed by *Policy*. As we require triple disjointness, $(s, a, s') \in \Delta$ is derived either from a individual rule or a pairwise one.

Let us assume that (s, a, s') is derived from an individual rule in which component i participates. Thanks to Lemma 2 and to *Policy*, we have that $marking(i, s_i) = marking(i, s'_i)$. As for all other components j we have $s_j = s'_j$, we end up with $Tks(sub, s) = Tks(sub, s')$.

Now we assume (s, a, s') is derived from a pairwise rule in which component i and j participate. Thanks to Lemma 1 and *Policy*, we have that $marking(i, s_i) + marking(j, s_j) = marking(i, s'_i) + marking(j, s'_j)$. As for all other components k we have $s_k = s'_k$, we end up with $Tks(sub, s) = Tks(sub, s')$. □

This technique should be particularly useful when applied to systems that implement a token-conservation mechanism to avoid reaching undesired states. We illustrate the application of this technique with Running Example 1.

Running example 1. FindMarkings($\mathcal{S}$) can result in $partitions = \{\{0, 1, 2\}\}$ and $marking = \{(0, s_0), (0, s_1), (1, s_1), (1, s_2), (2, s_1), (2, s_2)\}$; for conciseness, we represent a marking by the states that are assigned to true, so the missing states are assigned to false. With this information, we create the invariant $reach_C(s) \mathrel{\widehat{=}} Tks(s, \{0, 1, 2\}) = 1$. As we have that $Tks((s_1, s_2, s_2), \{0, 1, 2\}) = 3$, we have that this technique is able to prove that (s_1, s_2, s_2) is unreachable. This reachability invariant can show that this system can never be either filled with tokens, as in (s_1, s_2, s_2), or empty, as in (s_2, s_0, s_0). As these are the two cases in which this system is blocked, this technique can prove that $\mathcal{S}$ is deadlock-free. In this example, $\mathcal{S}$ is a token network with three components and a single token, initially held by L_0. This technique can, in fact, show that similar systems with N components and n (where $0 < n < N$) tokens are deadlock-free. □

4.2 Existential Technique

We term our second approach the *existential technique*. It enforces a token-flow policy where tokens can be created and destroyed but not eliminated altogether. We implement this new policy using the following definitions for enc_i and $enc_{i,j}$. For an individual transition (s, s') of component i, we define $enc_i(s, s')$ as follows. It says that such transitions can create but not destroy tokens.

$$enc_i(s, s') \mathrel{\widehat{=}} t_{i,s} \rightarrow t_{i,s'} \tag{3}$$

For a pairwise transition $(s, s') \mathrel{\widehat{=}} ((s_0, s_1), (s'_0, s'_1))$ involving components i and j, i and j can create or destroy tokens, provided that whenever a token is destroyed one of i and j continues to hold one. Thus the only way a token can be destroyed is in a pairwise transition where both parties hold a token before and

only one after. The auxiliary variables $hastk_{src}$ and $hastk_{tgt}$ represent whether a component holds a token in the source s and target s' states, respectively.

$$enc_{i,j}(s,s') \mathrel{\widehat{=}} hastk_{src} \leftrightarrow (t_{i,s_0} \vee t_{j,s_1}) \wedge hastk_{tgt} \leftrightarrow (t_{i,s'_0} \vee t_{j,s'_1}) \wedge hastk_{src} \leftrightarrow hastk_{tgt} \quad (4)$$

So, for this technique, we have the following SAT formula:

Definition 12. *For the supercombinator machine $\mathcal{S}$,*

$$\mathcal{F} \mathrel{\widehat{=}} Policy \wedge NotAlwaysHoldingToken \wedge Participation$$

where Policy uses enc_i and $enc_{i,j}$ as defined in (3) *and* (4)*, respectively.*

The existential technique uses FindMarkings presented in Algorithm 2 to systematically find markings. It works exactly like the one presented for the conservative technique except that it does a second minimisation step, carried out by FurtherMinimise. The functions Minimise and ExtractMarking are as described in Algorithm 1.

While Minimise tries to minimise the subsystem being marked, FurtherMinimise tries to minimise the timespan in which components hold a token. Given the minimal assignment found by Minimise, it tries to reduce the number of component states where tokens are held[1]. This second minimisation is an attempt to prevent the creation of spurious tokens; for instance, the creation of unnecessary tokens by individual transitions. Again, markings and subsystems marked are recorded in the global fields *marking* and *partitions*.

The information in *partitions* and *marking* is, once again, used to create reachability invariants. Note that our token-flow policy allows tokens to be destroyed as long as tokens are not completely annihilated from the system. So, as this technique guarantees that at least a token exists initially, a token should exists at all times. The reachability invariant that we propose enforce this *existential property* for each subsystem in *partitions*.

Definition 13. *Let $\mathcal{S} = (\langle L_1, \ldots, L_n \rangle, \mathcal{R})$ be a supercombinator machine where $\hat{s}_i$ is the starting state for L_i, partitions and marking the sets recorded after the execution of* FindMarkings($\mathcal{S}$) *in Algorithm 2, and $marking(i,s)$ yields 1 if the state s of component i is assigned to true, and 0 otherwise. Also, $Tks(sub,s) \mathrel{\widehat{=}} \sum_{i \in sub} marking(i,s_i)$. The reachability invariant $reach_E(s)$ is as follows:*

$$reach_E(s) \mathrel{\widehat{=}} \forall\, sub \in partitions \bullet Tks(sub,s) \geq 1$$

Lemma 4. $reachable(s) \Rightarrow reach_E(s)$

Proof. Let *sub* be a subsystem in *partitions*. For $\hat{s}$ the starting state of the LTS induced by system $\mathcal{S}$, we have by construction that $Tks(sub,\hat{s}) \geq 1$; Minimise ensures that at least one component in *sub* holds a token. Each transition preserves

[1] Setting the polarity of SAT variables, so that the solver first decides to assign variables to *false*, can substantially speed this minimisation process.

Algorithm 2 Algorithm to find existential token-structures

1: **function** FINDMARKINGS($\mathcal{S}$)
2: $partition := \emptyset$; $marking := \emptyset$
3: Construct $\mathcal{F}$ for $\mathcal{S}$
4: **while** SOLVE($\mathcal{F}$) **do**
5: MINIMISE($\mathcal{F}$)
6: FURTHERMINIMISE($\mathcal{F}$)
7: EXTRACTMARKING($\mathcal{A}$)
8: $\mathcal{F} := \mathcal{F} \wedge (\bigwedge_{i \in \{1 \ldots n\} \wedge \mathcal{A}(p_i)} \neg p_i)$
9: **end while**
10: **end function**

11: **function** FURTHERMINIMISE($\mathcal{F}$)
12: **repeat**
13: $\mathcal{F} := \mathcal{F} \wedge (\bigvee_{\substack{i \in \{1 \ldots n\} \wedge \mathcal{A}(p_i) \\ \wedge s \in S_i \wedge \mathcal{A}(t_{i,s})}} \neg t_{i,s}) \wedge (\bigwedge_{\substack{i : \{1 \ldots n\} \wedge \mathcal{A}(p_i) \\ \wedge s \in S_i \wedge \neg \mathcal{A}(t_{i,s})}} \neg t_{i,s})$
14: **until not** SOLVE($\mathcal{F}$)
15: **end function**

this invariant, that is, if $(s, a, s') \in \Delta$ and $Tks(sub, s) \geq 1$ then $Tks(sub, s') \geq 1$. This is guaranteed by *Policy*. Again, triple disjointness guarantees that $(s, a, s') \in \Delta$ is derived either from a individual rule or a pairwise one.

Let us assume that (s, a, s') is derived from an individual rule in which component i participates. Thanks to Lemma 2 and to *Policy*, we have that $marking(i, s'_i) \geq marking(i, s_i)$. As for all other components j we have $s_j = s'_j$, we end up with $Tks(sub, s') \geq Tks(sub, s)$. From $Tks(sub, s') \geq Tks(sub, s)$ and $Tks(sub, s) \geq 1$, we can conclude $Tks(sub, s') \geq 1$.

Now we assume (s, a, s') is derived from a pairwise rule in which component i and j participate. Thanks to Lemma 1 and *Policy*, we have that if $marking(i, s_i) + marking(j, s_j) \geq 1$ then $marking(i, s'_i) + marking(j, s'_j) \geq 1$. Assuming $marking(i, s_i) + marking(j, s_j) \geq 1$, we have $Tks(sub, s') \geq 1$. On the other hand, if $marking(i, s_i) + marking(j, s_j) = 0$, as we have $Tks(sub, s) \geq 1$, there must be a component $k \in sub$ such that $marking(k, s_k) = 1$. Hence, as for all other components k we have $s_k = s'_k$, we end up with $Tks(sub, s') \geq 1$. □

This technique should be particularly useful when applied to systems where tokens represent property of components and the fact that at least one component always has this property (i.e. a token) prevents the system from reaching a "bad" state. We illustrate the application this technique with Running Example 2.

Running example 2. Applying FINDMARKINGS to $\mathcal{S}$ can result in $partitions = \{\{0, 1, 2\}\}$ and

$$marking = \{(0, s_0), (0, s_1), (0, s_2), (0, s_3), (1, s_0), (1, s_1), (1, s_2), (1, s_3), (2, s_0), (2, s_1), (2, s_2), (2, s_3)\}$$

With this information, we create invariant $reach_E(s) \mathrel{\widehat{=}} Tks(s, \{0,1,2\}) \geq 1$. For this examples, we can interpret tokens as marking states in which the component is *not* full, and the invariant being that all components cannot be full at the same time. As we have that $Tks((s_6, s_6, s_6), \{0,1,2\}) = 0$, this technique is able to prove that (s_6, s_6, s_6) is unreachable. As this state is the only on in which the system is blocked, this technique can prove that $\mathcal{S}$ is deadlock-free. In this example, $\mathcal{S}$ is a token network with three components, each of them has a two-slot buffer to store messages. This technique can, in fact, show that similar systems with $N \geq 3$ components with b-slot buffers, where $b \geq 2$, are deadlock-free. □

5 Checking Deadlock-Freedom

In this section we combine Pair, a technique proposed in [3], with the new reachability tests presented in Section 4. In this new framework, a potential deadlock is a pair candidate that meets our new reachability invariants.

Definition 14. *Let $\mathcal{S}$ be a supercombinator machine and $(S, \Sigma, \Delta, \hat{s})$ its induced LTS. A state $s \in S$ is a* deadlock candidate *iff $deadlock\_candidate(s)$ holds, where $deadlock\_candidate(s) \mathrel{\widehat{=}} pair\_candidate(s) \wedge reach_C(s) \wedge reach_E(s)$.*

Given that our reachability tests over-approximate reachability and that every deadlock is also a Pair candidate [3], every deadlock must also be a deadlock candidate. So, a system free of deadlock candidates has to be deadlock free.

Theorem 1. *If a supercombinator machine is deadlock-candidate free, then it must also be deadlock free.*

Proof. This follows from the fact that every deadlock is also a Pair candidate [3], and Lemmas 3 and 4.

Our new characterisation is clearly more precise than the Pair one, but it remains imprecise: a blocked state can be unreachable and yet meet our two reachability invariants. Nevertheless, by conjoining these new reachability tests, we tighten the state space analysed. Observe that it only takes one failed test to consider a state unreachable. Furthermore, we note that the techniques presented in Section 4 might generate different reachability invariants for the same system. This means that we might have different outcomes when verifying systems with this deadlock-checking technique. We illustrate the unpredictability and incompleteness of our method with the following example.

Example 1. Let $\mathcal{S} = (\langle L_1, L_2, L_3 \rangle, \mathcal{R})$ be the supercombinator machine such that L_1, L_2 and L_3 are described in Figure 3 and $\mathcal{R}$ requires components to synchronise on shared events. The states (p_0, q_0, r_1) and (p_1, q_1, r_2) are blocked but not reachable, so neither of them represents a deadlock. Let us consider $partition = \{1,2,3\}$, $marking = \{p_1, q_1, r_0, r_2\}$ and $marking' = \{p_0, q_0, r_0, r_1\}$. For $\mathcal{S}$, the conservative technique cannot find any markings, while the existential technique might compute either *partition* and *marking* or *partition* and $marking'$. If it

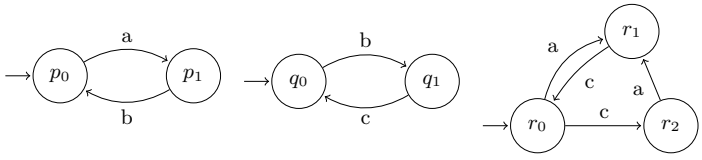

Fig. 3. LTSs of components L_1, L_2 and L_3, respectively.

computes *marking*, then (p_0, q_0, r_1) is proved unreachable but not (p_1, q_1, r_2). In case *marking*$'$ is computed, (p_1, q_1, r_2) is proved unreachable but not (p_0, q_0, r_1). As it cannot use *marking* and *marking*$'$ simultaneously, it cannot show that $\mathcal{S}$ is deadlock free. It could with a slightly modification in our techniques. □

5.1 Implementation

We built upon [3] to create an efficient implementation for our framework. So, we encode the search for a deadlock candidate as a satisfiability problem to be later checked by a SAT solver. For the remainder of this section, let $\mathcal{S} = (\langle L_1, \ldots, L_n \rangle, \mathcal{R})$, where $L_i = (S_i, \Sigma_i, \Delta_i, \hat{s}_i)$, be a supercombinator machine, and $(S, \Sigma, \Delta, \hat{s})$ its induced LTS.

In our propositional encoding, $st_{i,s}$ is the boolean variable representing the state s of component i. The assignment $st_{i,s} = true$ indicates this component state belongs to a deadlock candidate, whereas $st_{i,s} = false$ means it does not. Our formula $\mathcal{DC} \mathrel{\widehat{=}} PC \wedge Reach_C \wedge Reach_E$ is a conjunction of three sub-formulas, each of them captures a predicate of our deadlock characterisation. The combination of component states assigned to true in a satisfying assignment of $\mathcal{DC}$ forms a deadlock candidate.

The first sub-formula PC captures the pair-candidate characterisation; we reuse the propositional formula that is presented in [3]. The component states assigned to true in a satisfying assignment for PC form a Pair-candidate.

$Reach_C$ and $Reach_E$ capture the reachability invariants $reach_C$ and $reach_E$, respectively. To encode $Reach_x$ where x in $\{C, E\}$, we encode the markings with $Marking_x$ and the associated cardinality constraints with $Cardinality_x$. In the following, we assume $partitions_C$ and $marking_C$ were generated by our conservative technique, and $partitions_E$ and $marking_E$ by the existential one.

$Marking_x$, where x is C or E, uses a boolean variable tk_x^i for each component i (tk_x^i conveys whether component i holds a token) to encode the information recorded in $marking_x$, i.e. in which states components hold tokens.

$$Marking_x \mathrel{\widehat{=}} \bigwedge_{i \in \{1 \ldots n\} \wedge s \in S_i} st_{i,s} \rightarrow \begin{cases} tk_x^i & \textbf{if } (i, s, true) \in marking_x \\ \neg tk_x^i & \textbf{if } (i, s, false) \in marking_x \end{cases}$$

The cardinality constraint $Cardinality_C$ uses the variables tk_C^i to make sure that, in a satisfying assignment, subsystems in *partitions* have their expected number of tokens. Let *sub* be a subsystem in $partitions_C$, $\overline{tk}_C^{sub}$ the vector of

variables tk_C^i such that $i \in sub$, $\overline{x}^{sub}$ a vector of fresh boolean variable of size $|sub|$, and $N_C^{sub} = \sum_{i \in sub} marking_C(i, \hat{s}_i)$ the number of tokens confined in sub. Constraint $Sort(\overline{tk}_C^{sub}, \overline{x}^{sub})$ makes sure that $\overline{x}^{sub}$ is the result of sorting the values assigned to $\overline{tk}_C^{sub}$, i.e. true values come first. We use odd-even-merging sorting networks [7] to implement this sorting; they tend to provide a better compromise between the size of the encoding and the efficiency in which these constraints are checked [9]. Intuitively, $\overline{tk}_C^{sub}$ is a unary-unordered representation of the number of tokens being held by components in sub, whereas $\overline{x}^{sub}$ gives its unary-ordered representation. Constraint $Eq(\overline{x}^{sub}, N_C^{sub})$ ensures that $\overline{x}^{sub}$ is the unary-ordered representation of number N_C^{sub}.

$$Cardinality_C \mathrel{\widehat{=}} \bigwedge_{sub \in partitions_C} Sort(\overline{tk}_C^{sub}, \bar{x}_{sub}) \wedge Eq(\overline{x}_{sub}, N_C^{sub})$$

For instance, if in a satisfying assignment we have $\overline{tk}_C^{sub} = (true, false, true)$ (i.e. 101, a unary-unordered representation of 2), $Sort$ makes sure that $\overline{x}^{sub} = (true, true, false)$ (i.e. 110, *the* unary-ordered representation of 2).

The cardinality constraint $Cardinality_E$ uses the variables tk_E^i to ensure that, in a satisfying assignment, subsystems in *partitions* have at least one token. The "at least one token is being held" restriction is a trivial case of a cardinality constraint that can be implemented without need to sorting networks.

$$Cardinality_E \mathrel{\widehat{=}} \bigwedge_{sub \in partitions_E} (\bigvee_{i \in sub} tk_E^i)$$

6 Practical Evaluation

We here evaluate our new framework. FDR3's ability to analyse CSP and generate supercombinator machines is exploited in generating our SAT encoding, which is then checked by the Glucose 4.0 solver [6]. Our framework, implemented as the new DeadlOx-VT mode in the DeadlOx tool [4], detects both types of structures and combine them to prove deadlock-freedom[2]. A prototype of this tool and the models used in this section are available at [2]. For this experiment, we checked deadlock freedom for some CSP benchmark problems. The experiment was conducted on a dedicated machine with a quad-core Intel Core i5-4300U CPU @ 1.90GHz, and 8GB of RAM. We compare our prototype against: CSDD and FSDD (which are implemented in Martin's Deadlock Checker tool [15]); the original DeadlOx mode [4]; FDR3's built-in deadlock freedom assertion (FDR3) [10], and its combination with partial order reduction (FDR3p) [11] or compression techniques (FDR3c) [17]. We point out that only FDR3's techniques take advantage of the multicore setting, as our prototype is sequential.

[2] Note that the conditions for the conservative technique imply those for the existential one. So, a system that has a conservative invariant must have a existential one as well. We plan to improve our tool by, first, detecting conservative structures, and then only in case the invariants derived from these structures are not strong enough to prove deadlock-freedom, we would search for existential structures.

		Incomplete						Complete		
	N	DeadlOx	**DeadlOx-VT**			CSDD	FSDD	FDR3c	FDR3p	FDR3
			DF	**Co**	**Ex**					
DDB	5	0.14	**-**	**x**	**0.02**	-	-	0.31	0.18	0.15
	10	1.61	**-**	**x**	**0.47**	-	-	*	*	*
	20	57.75	**-**	**x**	**19.13**	-	-	*	*	*
Mat	10	3.68	**-**	**0.05**	**0.05**	0.17	-	15.52	0.29	*
	20	48.19	**-**	**0.37**	**0.30**	0.59	-	*	22.43	*
	30	*	**-**	**1.97**	**1.14**	2.08	-	*	*	*
Ring	500	0.80	**1.34**	**x**	**0.22**	-	0.86	0.64	*	*
	1000	2.38	**4.47**	**x**	**0.50**	-	2.63	1.49	*	*
	1500	5.02	**9.83**	**x**	**0.88**	-	5.93	6.68	*	*
Sched	500	0.55	**0.82**	**0.19**	**0.23**	0.45	-	3.05	103.26	*
	1000	1.29	**2.23**	**0.55**	**0.73**	0.84	-	8.72	*	*
	1500	2.29	**4.76**	**1.31**	**1.62**	1.30	-	20.06	*	*
Tk	50	0.78	**1.08**	**0.22**	**0.21**	-	-	+	45.62	11.85
	100	5.84	**7.40**	**1.43**	**1.35**	-	-	+	*	*
	200	66.44	**76.11**	**11.23**	**11.84**	-	-	+	*	*
Tk2	50	0.75	**1.05**	**0.21**	**0.21**	-	-	+	*	*
	100	5.71	**7.56**	**1.40**	**1.46**	-	-	+	*	*
	200	63.74	**79.13**	**12.62**	**12.91**	-	-	+	*	*
Tck	100	-	**0.48**	**0.09**	**0.09**	-	-	20.56	2.30	1.30
	200	-	**1.16**	**0.22**	**0.18**	-	-	209.96	23.84	9.24
	500	-	**4.85**	**0.67**	**0.58**	-	-	*	*	177.07
Tck2	100	-	**0.55**	**0.09**	**0.09**	-	-	20.66	*	*
	200	-	**1.24**	**0.21**	**0.22**	-	-	209.96	*	*
	500	-	**5.03**	**0.66**	**0.54**	-	-	*	*	*
RC	30	-	**18.81**	**4.59**	**4.50**	-	-	+	*	5.65
	40	-	**79.52**	**19.00**	**18.61**	-	-	+	*	36.05
	50	-	**241.36**	**54.97**	**54.69**	-	-	+	*	134.58
RC2	30	-	**19.08**	**4.66**	**4.52**	-	-	+	*	*
	40	-	**79.95**	**19.05**	**19.15**	-	-	+	*	*
	50	-	**243.39**	**55.88**	**55.58**	-	-	+	*	*
RE	25	-	**0.80**	**x**	**0.21**	-	-	+	*	*
	50	-	**5.21**	**x**	**1.42**	-	-	+	*	*
	100	-	**38.86**	**x**	**10.64**	-	-	+	*	*
RE10	30	-	**20.18**	**x**	**5.94**	-	-	+	*	*
	40	-	**44.93**	**x**	**13.09**	-	-	+	*	*
	50	-	**87.53**	**x**	**26.30**	-	-	+	*	*

Table 1. Benchmark efficiency comparison. N is a parameter that is used to alter the size of the system. We measure in seconds the time taken to check deadlock freedom for each system. * means that the method took longer than 300 seconds. - means that the method is unable to prove deadlock freedom. + means that no efficient compression technique could be found. For the DeadlOx-VT, we present the total time taken to verify deadlock-freedom in column DF, whereas columns Co and Ex present the time taken for token-structure detection by the conservative and existential techniques, respectively, and x means that a token-structure has not been detected. There was no significant difference between the time taken by successful and failed detections of token structures.

We analyse 12 systems that are deadlock free, triple disjoint and cannot be proved deadlock-free by pure local analysis. We evaluate systems that cannot be proved deadlock free by pure local analysis as we want to evaluate how well incomplete techniques can leverage global invariants. Out of these systems, 10 can be proved deadlock free by DeadlOx-VT, 6 by DeadlOx, 2 by CSDD, and 1 by FSDD. The systems that we evaluated are: a distributed database (DDB), a matrix multiplication system (Mat), a non-fillable ring system (Ring), Milner's scheduler (Sched), a token ring system with a single token (Tk), a token ring system with $N/2$ tokens (Tk2), a train track system with two trains (Tck), a train track system with $2N$ trains (Tck2), and four routing networks: RC and RC2 implement a conservative token mechanism with two and $N/2$ tokens intially, respectively, whereas RE and RE10 implement an existential token structure with components that have two-slot and 10-slot buffers, respectively. Table 1 presents the results that we obtain for them.

Our results attest that DeadlOx-VT is able to handle a class of systems that is different from the one tackled by the original DeadlOx, while faring similarly in terms of analysis time. Comparing to the complete approaches, incomplete frameworks are consistently faster than the best complete approach, which is the combination of FDR3's deadlock assertion with compression techniques, while being able to prove deadlock freedom for all the benchmark problems. We point out, however, that the effective use of compression techniques requires a careful and skilful application of those, whereas our method is fully automatic.

7 Conclusion

Motivated by deadlock analysis, we have demonstrated that token structures of concurrent systems, sometimes too subtle to be obviously recognisable as such, can be recognised by SAT checkers and used to prove safety properties of the system concerned. We have identified two types of token structures: the first one makes sure that tokens are conserved, and the second one ensures at least one token is present in the system at all times. While we have interpreted these structures as token mechanisms, there might be other views to them. For instance, as we discussed in the application of our existential technique to Running Example 2, tokens can be seen as the component property "component is not full".

In the case of deadlock checking, our token-structure-detection techniques can be used to create not only a useful framework for deadlock-freedom analysis but one that improves the precision of current incomplete locally-based frameworks. Our experiments have confirmed the usefulness of our new framework, showing deadlock freedom for classes of systems our previous local analysis tools could not handle. They have also demonstrated that, for the systems analysed, the SAT calculations used to detect token structures can be carried out efficiently.

There is nothing CSP-specific in our methods, other than that we have a systems described as a pairwise interacting LTSs. So, the ideas in this paper should transfer easily to any formalism where systems are described as such. DeadlOx-VT uses FDR3 to obtain supercombinator machines from systems

described using CSP, but an analogous tool could be created for other notations by replacing its use of FDR3 to generate such machines.

This work begs a number of questions. What other uses, besides deadlock-checking, do the types of invariant we have identified have? What other sorts of invariants are there where partitioning of component states can be efficiently calculated? An obvious one is to handle token systems where nodes can have more than one token, or where there are multiple tokens with different properties. We will aim to answer these questions in future research.

Acknowledgments The first author is a CAPES Foundation scholarship holder (Process no: 13201/13-1). The second and third authors are partially sponsored by DARPA under agreement number FA8750-12-2-0247 and EPSRC under agreement number EP/N022777.

References

1. Tilak Agerwala and Y-C Choed-Amphai. A synthesis rule for concurrent systems. In *Design Automation, 1978. 15th Conference on*, pages 305–311. IEEE, 1978.
2. Pedro Antonino, Thomas Gibson-Robinson, and A. W. Roscoe. Experiment package, 2016. `http://www.cs.ox.ac.uk/people/pedro.antonino/pkg-vt.zip`.
3. Pedro Antonino, Thomas Gibson-Robinson, and A.W. Roscoe. Efficient deadlock-freedom checking using local analysis and SAT solving. In *IFM*, number 9681 in LNCS, pages 345–360. Springer, 2016.
4. Pedro Antonino, Thomas Gibson-Robinson, and A.W. Roscoe. Tighter reachability criteria for deadlock freedom analysis. 2016.
5. Krzysztof R. Apt, Nissim Francez, and Willem P. De Roever. A proof system for communicating sequential processes. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 2(3):359–385, 1980.
6. Gilles Audemard and Laurent Simon. Predicting Learnt Clauses Quality in Modern SAT Solvers. IJCAI'09, pages 399–404, San Francisco, CA, USA, 2009.
7. K. E. Batcher. Sorting networks and their applications. In *Proceedings of the April 30–May 2, 1968, Spring Joint Computer Conference*, AFIPS '68 (Spring), pages 307–314, New York, NY, USA, 1968. ACM.
8. Saddek Bensalem and Yassine Lakhnech. Automatic Generation of Invariants. *Form. Methods Syst. Des.*, 15(1):75–92, July 1999.
9. Niklas Eén and Niklas Sörensson. Translating pseudo-boolean constraints into SAT. *JSAT*, 2(1-4):1–26, 2006.
10. Thomas Gibson-Robinson, Philip Armstrong, Alexandre Boulgakov, and A.W. Roscoe. FDR3 — A Modern Refinement Checker for CSP. In *TACAS*, volume 8413 of *LNCS*, pages 187–201, 2014.
11. Thomas Gibson-Robinson, Henri Hansen, A.W. Roscoe, and Xu Wang. Practical partial order reduction for CSP. In *NFM*, volume 9058 of *LNCS*, pages 188–203. Springer International Publishing, 2015.
12. C. A. R. Hoare. *Communicating Sequential Processes*. Prentice-Hall, 1985.
13. Leslie Lamport. Proving the correctness of multiprocess programs. *Software Engineering, IEEE Transactions on*, (2):125–143, 1977.
14. Jeremy M. R. Martin. *The Design and Construction of Deadlock-Free Concurrent Systems*. PhD thesis, University of Buckingham, 1996.

15. J.M.R. Martin and S.A. Jassim. An efficient technique for deadlock analysis of large scale process networks. In *FME '97*, pages 418–441, 1997.
16. T. Murata. Petri nets: Properties, analysis and applications. *Proceedings of the IEEE*, 77(4):541–580, Apr 1989.
17. A. W. Roscoe, Paul H. B. Gardiner, Michael Goldsmith, J. R. Hulance, D. M. Jackson, and J. B. Scattergood. Hierarchical compression for model-checking CSP or how to check 10^{20} dining philosophers for deadlock. In *TACAS*, pages 133–152, 1995.
18. A.W. Roscoe. *Understanding Concurrent Systems*. Springer, 2010.