SOFTWARE TESTING, VERIFICATION AND RELIABILITY
Softw. Test. Verif. Reliab. 0000; **00**:1–28
Published online in Wiley InterScience (www.interscience.wiley.com). DOI: 10.1002/stvr

Past-Free[ze] Reachability Analysis. Reaching further with DAG-directed exhaustive state-space analysis

Ciprian Teodorov*, Luka Le Roux, Zoé Drey and Philippe Dhaussy

Lab-STICC, UEB, ENSTA Bretagne, France

SUMMARY

Model-checking enables the automated formal verification of software systems through the explicit enumeration of all the reachable states. While this technique has been successfully applied to industrial systems, it suffers from the state-space explosion problem due to the exponential growth in the number of states with respect to the number of interacting components.
In this paper, we present a new reachability analysis algorithm, named Past-Free[ze], that reduces the state-space explosion problem by freeing parts of the state-space from memory. This algorithm relies on the explicit isolation of the acyclic parts of the system before analysis. The parallel composition of these parts drives the reachability analysis, the core of all model-checkers. During the execution, the past states of the system are freed from memory making room for more future states. To enable counter-example construction the past states can be stored on external storage.
To show the effectiveness of the approach the algorithm was implemented in the OBP *Observation Engine* and was evaluated both on a synthetic benchmark and on realistic case studies from automotive and aerospace domains. The benchmark, composed of 50 test cases, shows that in average 75% of the state-space can be dropped from memory thus enabling the exploration of up to 14 times more states than traditional approaches. Moreover, in some cases the reachability analysis time can be reduced by up to 25%. In realistic settings, the use of Past-Free[ze] enabled the exploration of a state-space 4.5 times larger on the automotive case study, where almost 50% of the states are freed from memory. Moreover, this approach offers the possibility of analyzing an arbitrary number of interactions between the environment and the system-under-verification; for instance, in the case of the aerospace example 1000 pilot/system interactions could be analyzed unraveling an 80GB state-space using only 10GB of memory. Copyright © 0000 John Wiley & Sons, Ltd.

Received . . .

KEY WORDS: Reachability analysis, formal verification, context-aware verification, directed-acyclic graph, semi-external algorithms

1. INTRODUCTION

Since its introduction in the early 1980s, model-checking [12, 36] provides an automated formal approach for the verification of complex requirements of hardware and software systems. This technique relies on the exhaustive analysis of all states in the system to check if it correctly implements the specifications, usually expressed using temporal logics. However, because of the internal complexity of the studied systems, model-checking is often challenged with unmanageable large state-space, a problem known as the state-space explosion problem [13, 33]. Numerous techniques, such as symbolic model-checking [10] and partial-order reduction [41], have been

*Correspondence to: ENSTA Bretagne, 2 rue François Verny, Brest, France. Email: ciprian.teodorov@ensta-bretagne.fr

proposed to reduce the impact of this problem effectively pushing the inherent limits of model-checking further and further. Complementary to these techniques are approaches based on the specification of environments relevant to the studied system [32, 40, 43]. These approaches propose tools that generate environments, based either on assumptions on the system and its interactions with the environment [32, 40], or on the properties that need to be verified [43].

This paper directly addresses the state-space explosion problem, leveraging the isolation of acyclic behaviors in the system specification. Specifically, we propose a novel exhaustive analysis algorithm that reduces the memory consumption by using the external storage to store the "past-states" of the system-under-study (SUS). This algorithm, named Past-Free[ze], relies on the isolation of the acyclic components of the SUS, which are used to drive the reachability analysis. The graph induced by these acyclic behaviors identifies "clusters of states" that can be freed from memory and saved to disk for later error reporting such as counter-example extraction.

To show the effectiveness of our technique, the Past-Free[ze] algorithm was implemented in the OBP *Observation Engine*, a model-checking tool. This tool integrates the Context-aware Verification (CaV) methodology [20] that isolates the system specification from its environment. For the environment specification the CaV methodology uses a dedicated language, named CDL [21], which enforces an acyclic interaction scenario invariant. This implementation of the Past-Free[ze] algorithm was evaluated on a benchmark of 50 test cases (Sec. 4.3), and the results were compared with the standard reachability algorithms implemented in SPIN [29] and OBP (Sec. 4.4). The Past-Free[ze] algorithm enabled the analysis of systems with up to 14 times more states. Moreover, through the use of the "clusters of states" induced by the acyclic component the run-time of the analysis can be drastically reduced (Sec 4.5).

The evaluation of our approach on two realistic case studies from the automotive and aerospace industry shows that Past-Free[ze] algorithm is effective in a practical setting, enabling the exploration of larger state-spaces than the baseline implementations in SPIN [29] and OBP *Observation Engine*. Moreover, when coupled with the automatic-context splitting technique [18], for the automotive case study it enabled the analysis of a state-space 4.78 times larger and for the aerospace case study it enabled the analysis of arbitrarily long interaction scenarios without increasing the amount of physical memory needed nor the analysis time.

The main contributions of this paper are:

- *A new algorithm for reachability analysis* : this algorithm leverages the acyclic part of a system and pushes further the limits of the reachability analysis by freeing reached states from memory, and saving them to disk for later error reporting.

- *An integration of our algorithm with a model-checking tool* : the practical use of this algorithm is shown by its integration into the OBP *Observation Engine* toolset.

- *A quantitative evaluation* : a benchmark of 50 case studies of transition systems with acyclic behavior has been evaluated, and the Past-Free[ze]-based exploration technique has been compared to reachability results with SPIN & OBP.

- *A realistic case-study experiment* : through the integration with the Context-aware Verification approach, the Past-Free[ze] technique has been used to analyze two real-size case studies from automotive and aerospace industry.

Section 2 describes our main contribution, the Past-Free[ze] algorithm and introduces three evaluation metrics. Section 3 introduces the CaV methodology that offers a methodological framework for the specification of acyclic behaviors, which enables the effective use of Past-Free[ze] in a realistic setting. Section 4 presents the evaluation of Past-Free[ze] with respect to other reachability algorithms. Section 5 illustrates the effectiveness of the Past-Free[ze] algorithm on two realistic case-studies. In Section 6 we overview the state-of-the-art emphasizing the advantages and the complementarity of our approach with existing model-checking strategies. Section 7 concludes this study providing some future research directions.

Prepared using stvrauth.cls
DOI: 10.1002/stvr

2. PAST-FREE[ZE] REACHABILITY ANALYSIS

In this section, the core of our contribution is presented. The Past-Free[ze] algorithm exploits the acyclic behaviors for reducing the memory pressure during the reachability analysis.

2.1. Reachability & Model-checking

When verifying properties, through explicit-state model checking, all the possible behaviors of the system are enumerated and the properties are checked.

In the model-checking literature the theoretical framework, used for capturing the system semantics, is typically restricted to either Kripke structures or Labeled Transition Systems (LTS), which are the two most prominent models used in concurrency theory. These two graph-based models are commonly believed to be equi-expressive [37], the main difference being the labeling strategy: Kripke structures being state-based (states are labeled) while Labeled Transition Systems are transition-based (transitions are labeled). For the purposes of this study we will use the LTS model.

Formally, a LTS T is a 4-tuple $\langle S, I, Act, \rightarrow \rangle$ with:

- S is a set of states;
- I is the set of initial states: $I \subseteq S$;
- Act is a set of actions;
- $\rightarrow \subseteq S \times Act \times S$ is a transition relation[†], we will also use $\mathbb{T} \subseteq S \times Act \times S$ when referring to the set of transitions of the LTS;

In some approaches that rely on sophisticated state-space encoding techniques, such as BDDs for symbolic model-checking [10], the concept of LTS (or Kripke structure) is present only for theoretical reasons. In the case of explicit-state model-checking the LTS is also used at the implementation level to represent the state-space. In this case the LTS is constructed by unraveling the sequential behaviors of the system (LTS themselves) composed using either synchronous ($|$) or asynchronous[‡] ($||$) operators. This step is known as the *reachability analysis* of the system. Regarding the property verification, two cases can be further identified: *a*) online model-checking, where the properties are verified on-the-fly while constructing the LTS; *b*) and offline model-checking, in which case, firstly the LTS is constructed and the properties are verified on the result.

During the *reachability analysis*, due to the exponential growth of the number of system-states relative to the number of interacting components, most of the time the number of LTS states (also further referred to as reachable configurations) is too large to be contained in memory. This exponential growth, also known as the *state-space explosion*, represents the main challenge of the model-checking technique.

In this study, we directly address this challenge focusing on the *reachability analysis* of asynchronously-composed finite LTS systems. It should be noted that the LTS model can equally represent the interacting behaviors before composition and the composition results (after reachability analysis). In the following sections we mostly use a graph-theoretic interpretation of LTS for capturing the structural properties of the interacting behaviors **before** composition. This should not be confused with the post-processing analysis techniques using the reachability results (the LTS **after** the composition), like checking temporal logic formulas (such as LTL).

2.2. Acyclic LTS

To tackle the *state-space explosion* challenge we propose the identification and explicit isolation of structurally different LTS behaviors, such that before the reachability analysis the system is decomposed into its cyclic (T_c) and acyclic (T_a) LTSs, which are then composed under the $||$

[†]Sometimes $\rightarrow$ is considered a total relation and Act is extended with $\{\tau\}$ – an internal hidden action
[‡]The asynchronous composition is sometimes named the interleaving semantics of concurrency

Prepared using stvrauth.cls
DOI: 10.1002/stvr

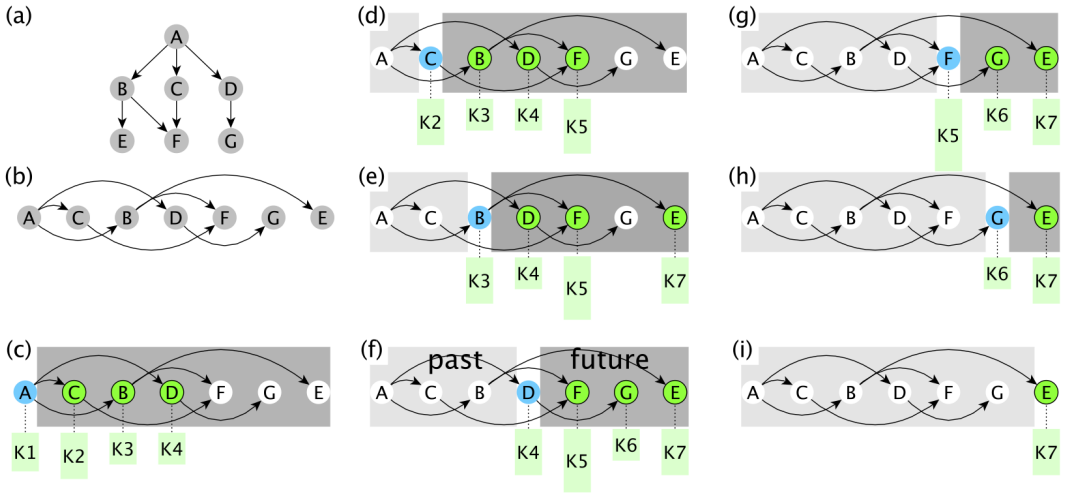


Figure 1. Illustration of the Past-Free[ze] reachability algorithm. (a) Acyclic LTS T_a; (b) Topological ordering of T_a; (c)-(i) Reachability analysis of $T_c||T_a$ using the Past-Free[ze] algorithm. The K1-K7 rectangles represent the sets of configurations (K_i), their different heights emphasizing the different number of configurations induced by each state of T_a. The white nodes represent either past configuration clusters (freed from memory) or future configuration clusters not yet reachable (not present in memory). Blue nodes between the shaded areas represent the cluster being analyzed, while the green nodes with black circles around are live clusters present in memory (either reached in the past, or currently reachable).

operator. This partitioning can be obtained using the LTS-induced graph. An LTS-induced graph $(G(V,E),\ S = V \wedge \mathbb{T} = E)$ is the graph-theoretic view of an LTS where the states are seen as vertices (V) and the transitions are seen as edges (E). A path in this graph, representing an execution trace, is defined as a sequence of states $(s_i,\ldots,s_j),\ \forall s_i,s_j \in S,\ with\ i,j \in \mathbb{N}^+$ connected by $j-i$ transitions from $\mathbb{T}$. In this case, an acyclic LTS, also referred to as a directed-acyclic graph (DAG), does not contain any cyclic path $(s_i,\ldots,s_i), \forall s_i \in S,\ i \in \mathbb{N}^+$. Without loss of generality, in the following, we will consider only one acyclic LTS behavior T_a, knowing that the acyclicity is conserved by the parallel composition operator[§].

Conceptually, an acyclic LTS can be viewed as a set (exhaustive or not) of execution scenarios to be analysed. In practice, an acyclic LTS can represent a set of test scenarios, an unfolding of a system partition, or the unrolling of the execution environment. For the purpose of this study, we use the Context-aware Verification [17, 18], a tool supported methodology presented in Section 3, for the explicit identification and the specification of the acyclic LTS. This paper focuses on using the acyclic part of the system to drive the whole reachability analysis process, independently of the (possible) decomposition of the state-space that can be achieved through the exploitation of the acyclic LTS.

Our technique relies on the observation that *acyclic graphs can be ordered* such that when considering a given vertex in this ordering all its predecessors were considered before. This ordering is known as the *topological ordering* of a DAG [15], and can be formally defined as a linear order between the vertices of a DAG (the states of the acyclic LTS in our case) such that if there exists a transition $u \rightarrow v$ between two states $u,v \in S$ then u is present before v in the ordering, expressed as $u < v$ (u precedes v, or u is an ancestor of v). Figure 1(b) shows one topological order of the DAG in Figure 1(a). Such an ordering can be constructed in linear time with respect to the size of the DAG, in our case the complexity being $\mathcal{O}(|S| + |\mathbb{T}|)$, where $|\mathbb{X}|$ represents the cardinality of the set $\mathbb{X}$ [15].

[§]In general, one acyclic LTS can be obtained by the composition of all acyclic behaviors in the system

DOI: 10.1002/stvr

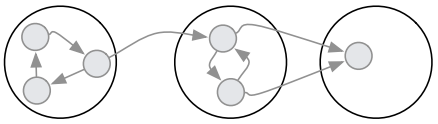


Figure 2. Clustering of an acyclic LTS

Relying on the *topological ordering* the reachability algorithm can "*start forgetting the past to focus on the future*". Practically this means that if the LTS states are indexed according to the ordering, the reachability routine can then consider all states at a particular position i before considering any future states. Moreover, when passing to the next state $i+1$, all past "states" (including i) can be freed from memory, since they are all already processed and the analysis never goes back (there are no cycles). Thus, this technique effectively reduces the memory requirements during reachability analysis enabling the exhaustive exploration of larger systems. The details of this reachability analysis algorithm, named Past-Free[ze], are discussed in the next section.

2.3. Past-Free[ze] reachability algorithm

In the following we consider a system Sys composed of a cyclic LTS T_c and an acyclic LTS T_a, both of them being finite. Let the reachable state-space for the parallel composition of all processes in T_c be $S_c \subseteq \{(s_1, \ldots, s_n) | s_i \in S_c^i\}$, where S_c^i represents the state space of each process i in T_c. For T_a we map its state-space to values from $\mathbb{N}^+$. Thus, $S_a = \{i | i \in [1, n]\}$, where n is the number of states of the acyclic process. Then let $\boldsymbol{\mathcal{S}} \subseteq \{\langle s, i\rangle | s = (s_1, \ldots, s_n) \wedge s_k \in S_c^k \wedge i \in S_a\}$ be the reachable state-space for the system Sys under the parallel composition operator $T_a || T_c$. For brevity in the following we will use a "don't care" operand in the state notation, such that for an arbitrary value i, $\langle -, i\rangle = \{\langle s, i\rangle | s \in S_c\}$. Moreover, in the following a *configuration* will refer to a composed state $\langle s, i\rangle$, while a *state* will refer to either the s (or i) component of a configuration.

As stated in the previous section, the Past-Free[ze] algorithm relies on the observation that if the reachability analysis processes the reachable configurations in the topological order induced by the acyclic component then the analysis never goes back to a configuration found in a previously analyzed cluster. This observation is formalized in the following Theorem:

Theorem 1
If S_a is topologically sorted, and the reachability analysis processes the configurations $\langle -, -\rangle \in \boldsymbol{\mathcal{S}}$ in the resulting total order; then, once all configurations $\langle -, i\rangle, \forall i \in S_a$ are considered, the reachability analysis of the composition $S_c || S_a$ will never reach a configuration $\langle -, j\rangle$, where $j \leq i$.

Proof
Suppose not. Suppose that the reachability analysis of $S_c || S_a$ reaches a configuration $\langle -, j\rangle$ after processing all configurations $\langle -, i\rangle$, with $j \leq i$ then:
– if $j = i \Rightarrow \exists \langle -, i\rangle \in \boldsymbol{\mathcal{S}}$ such that $\langle -, i\rangle$ was not analyzed when considering all configurations $\langle -, i\rangle$. (Contradiction!)
– if $j < i \Rightarrow$ since T_a progresses only by executing a transition from $\mathbb{T}_a, \exists$ an edge $\langle j, i\rangle \in T_a$ such that the vertex j is an ancestor of vertex i in the DAG. The topological sort places all ancestors x of a vertex i before it in the partial order, hence j is before i. Thus, since all configurations S are processes in topological order it follows that $\langle -, j\rangle$ was analyzed before $\langle -, i\rangle$. (Contradiction!) □

The acyclic component T_a induces a clustering of the space of configurations as sketched in Figure 2. Each cluster i is identified by the state of T_a and contains the set of configurations $\langle -, i\rangle$. Our technique relies on this clustering to easily identify, at the beginning of the analysis of any cluster i, all sets of configurations $\langle -, k\rangle$, with $k < i$, which can be freed from memory.

To show the intuition behind our approach, let us take for example the LTS-induced DAG in Figure 1(a) composed with an arbitrary cyclic LTS (T_c). One possible topological ordering of T_a is shown in Figure 1(b). Using this ordering the reachability analysis (Figure 1(c)) starts by processing all configurations reachable from the initial state $\langle s_0, A\rangle$. When a transition from T_c is fired the

Prepared using stvrauth.cls
DOI: 10.1002/stvr

resulting configurations will be elements of $\langle -, A\rangle$. When a transition from T_a is fired the resulting configuration will be in $\langle -, C\rangle$, $\langle -, B\rangle$, or $\langle -, D\rangle$. Once all states in $\langle -, A\rangle$ are processed, the analysis moves to the next cluster($\langle -, C\rangle$ in our case), and the previous cluster $\langle -, A\rangle$ is stored on disk and freed from memory (see Figure 1(d)). This process repeats until the last configuration from cluster $\langle -, E\rangle$ is analyzed, at which point the analysis ends (see Figure 1(i)).

The Past-Free[ze] algorithm, schematically presented in Algorithm 1, relies on a *priority queue* data-structure Q for storing the reached configurations before analysis. This priority queue orders its elements in increasing order based on the total order defined by the topological sorted states of T_a. In a typical setting, the reachability algorithms use another *set* data-structure for storing the known configurations to prevent multiple analysis in the case of loops in the resulting LTS. In our case we replaced this set by a list of sets with $|S_a|$ elements so that the clusters induced by the acyclic LTS (T_a) are accessible in constant time.

Algorithm 1 Past-Free[ze] context-aware reachability analysis algorithm

function PAST-FREE[ZE]REACHABILITY(T_a, T_c)
  sort all states S_a of T_a in **topological order**
  for all $i \in [0, |S_a|)$ **do**
    let the **set** of configurations $K_i \leftarrow \emptyset$
  end for
  priority queue of configurations $Q \leftarrow \{\langle s_0, 0\rangle\}$
  let $id \leftarrow 0$
  while $Q \neq \emptyset$ **do**
    let $\langle s, i\rangle \leftarrow min(Q)$
    if $id \neq i$ **then**
      $IO.saveAll(K_{id})$ // **save** the cluster to disk (to enable counter-example extraction)
      $K_{id} \leftarrow \emptyset$ // **free** all configurations in K_{id}
      $id \leftarrow i$
    end if
    if $\exists\langle t, j\rangle$ such that $\langle s, i\rangle \rightarrow \langle t, j\rangle$ not fired **then**
      let $\langle t, j\rangle$ be such a configuration
      fire transition $\langle s, i\rangle \rightarrow \langle t, j\rangle$
      if $\langle t, j\rangle \notin K_j$ **then**
        $Q \leftarrow Q \cup \{\langle t, j\rangle\}$
        $K_j \leftarrow K_j \cup \{\langle t, j\rangle\}$
      end if
    else
      $Q \leftarrow Q \setminus \{\langle s, i\rangle\}$ – delete-min(Q)
    end if
  end while
  $IO.saveAll(K_{id})$
  return $IO.getReferenceToResults$
end function

The *Past-Free[ze]Reachability* function, in Algorithm 1, takes the T_a and T_c as inputs, it sorts the states of T_a in topological order, then it initializes all K_i sets to the empty set, and inserts the initial configuration $\langle s_0, 0\rangle$ in the priority queue Q. The id variable, initially set to 0 (the initial state id $\in S_a$), is used to keep track of the progress with respect to T_a. The main loop executes until there are no more configurations to be analyzed (Q is empty). This loop starts by retrieving the configuration $\langle s, i\rangle$ with the smallest i (according to the ordering) from Q. If the index of the configuration changed ($id \neq i$) then all configurations with index id were analyzed, thus they can be stored on disk ($IO.saveAll$) and freed from memory ($K_{id} \leftarrow \emptyset$) while updating id. Then the algorithm progresses by firing a transition starting from $\langle s, i\rangle$ that was not fired yet. If the resulting configuration $\langle t, j\rangle$ is new (not in the cluster K_j) it is inserted into Q for future analysis and in

DOI: 10.1002/stvr

K_j for bookkeeping. When all transitions from $\langle s, i\rangle$ have been fired the configuration is removed from Q, and the analysis of a new configuration starts. At the end of the loop, the last cluster of configurations (with index $= |S_a| - 1$) is stored on disk, and a reference to the results is returned. The clusters stored in the disk serve to later error reporting, e.g., in the case where a counter example needs to be provided to the system designer.

Compared to depth-first or breadth-first (BFS) state-space analysis strategies, the asymptotic complexity of the Past-Free[ze] algorithm is increased mainly due to the need to iterate over all configurations in $\boldsymbol{\mathcal{S}}$ to free the memory (and save the configurations on disk). Moreover, since the FIFO/LIFO queue Q is replaced by a priority-queue implemented using a balanced binary heap, the element deletion complexity (delete-min) is increased from a constant factor to $\mathcal{O}(log|Q|)$. However, the usage of multiple sets for storing the configuration clusters instead of one single set reduces the probability of hash-collisions by a factor equal to the number of states in the acyclic LTS ($|S_a|$). Hence in practice, if the IO overhead is not considered, the Past-Free[ze] algorithm is generally faster than a BFS-based algorithm, as shown in Sec. 4.5(Figure 10).

General case and the composition order. The Algorithm 1 presents the algorithm for the parallel composition of a cyclic LTS T_c and an acyclic LTS T_a.

In the general case, observe that, the presence of multiple components in T_c does not affect the behavior of Algorithm 1, $\langle s, i\rangle$ will simply become $\langle\langle s_1^c, \ldots, s_n^c\rangle, i\rangle$. In other words, the PastFree[ze] algorithm does not require to compute the composition of the cyclic components beforehand.

In the case of multiple acyclic components, however, their composition should be realized before applying PastFree[ze] to enable the computation of the topological order. Nevertheless, if amongst the acyclic components, only one is used to drive the reachability (the T_a component) the others can be considered as components of T_c.

Consider, for instance, an arbitrary system composed of a set of n cyclic ($\boldsymbol{\mathcal{T}}_c = \{T_c^i | i \in [1 \ldots n]\}$) and a set of m acyclic ($\boldsymbol{\mathcal{T}}_a = \{T_a^i | i \in [1 \ldots m]\}$) components. Let $\boldsymbol{\mathcal{A}}$ be a non-empty subset of $\boldsymbol{\mathcal{T}}_a$. Using PastFree[ze], the reachability of this system (T_{result}) can be computed as follows:

$$T_a \xrightarrow{Reachability} T_1 || \cdots || T_k, where\ k = |\boldsymbol{\mathcal{A}}|\ and\ T_i \in \boldsymbol{\mathcal{A}}, \tag{1a}$$

$$T_{result} \xrightarrow{Past-Free[ze]} T_a \,||\, (T_c^1 \,||\cdots||\, T_c^m)||(T_a^1||\cdots||T_a^k), where\ T_a^i \in \boldsymbol{\mathcal{T}}_a \setminus \boldsymbol{\mathcal{A}} \tag{1b}$$

The composition of the acyclic components in $\boldsymbol{\mathcal{A}}$ (Equation 1a) is performed using an arbitrary reachability strategy (DFS for instance), hence obtaining an acyclic state-space T_a, which can be topologically sorted. The result of this preliminary composition step is injected in the PastFree[ze] algorithm along with all the cyclic components ($T_c^1 \,||\cdots||\, T_c^m$) and the remaining acyclic components ($T_a^1||\cdots||T_a^k$, which are in $\boldsymbol{\mathcal{T}}_a \setminus \boldsymbol{\mathcal{A}}$). Note that the k elements left in $\boldsymbol{\mathcal{T}}_a \setminus \boldsymbol{\mathcal{A}}$ are composed during the PastFree[ze] run (Equation 1b) and are extending the "cyclic" tuple of the global configuration, which becomes $\langle\langle s_1^c, \ldots, s_n^c, s_1^a, \ldots, s_k^a\rangle, i\rangle$.

It should be noted that besides enabling the topological ordering, precomposing the acyclic LTSs reduces the size of each configuration in the state-space. The otherwise composite tuple $\langle s_1^a, \ldots, s_k^a\rangle$ (where s_i^a represent the state of the LTS T_i in $\boldsymbol{\mathcal{A}}$) is replaced by the index $i \in [1..|T_a|]$ computed by the topological sort. Moreover, in the worst case (if only one acyclic component is included in $\boldsymbol{\mathcal{A}}$) the size of the configuration tuple is equal to the number of components in the system ($m + n$).

State matching. As a direct consequence of Theorem 1 during the reachability algorithm only forward transitions would occur, the target configurations, in these cases, would be directly matched by using an in-memory equality operator. However, for some types of offline analysis there might be a need for considering different equivalence relations between the reachable states. These equivalence relations could potentially abstract over the acyclic LTS, and thus, obtain arbitrary relations between the clusters induced by the acyclic LTS. These cases would impose the definition of an adequate I/O efficient state-matching strategy but for the purpose of this study they are seen as orthogonal to the reachability problem.

DOI: 10.1002/stvr

2.4. Metrics

To better analyze the characteristics of our approach in the following paragraphs three metrics are introduced: the *reached-future*, the *freezability*, and the *relative-progress*. In Sec. 4 these will be used in conjunction with more traditional metrics (such as the number of states and transitions explored, the run-time, etc.) to interpret the experimental results.

Reached-Future Metric This metric, $|\boldsymbol{\mathcal{R}}|_{max}$, represents a maximum bound on the total number of sets K_i that are present in memory at any moment during the reachability analysis, and it can be computed analytically using the following formulas:

$$\begin{aligned} \boldsymbol{\mathcal{R}}_i &= \left(\bigcup_{j=1}^{i} fanout(i) \right) \setminus \{k |\ k \in [1,i]\}, \forall i \in S_a \\ |\boldsymbol{\mathcal{R}}|_{max} &= 1 + Max_{i=1}^{|S_a|}(|\boldsymbol{\mathcal{R}}_i|) \end{aligned} \tag{2}$$

where $fanout(i)$ represents the set of states reached from the i^{th} state of the acyclic LTS. In other words, for a state i, $\boldsymbol{\mathcal{R}}_i$ represents the set of states after i (in the topological sort) that are reached either before or from i. Hence, the maximum number of clusters present in memory during the exploration is equal to the maximum $\boldsymbol{\mathcal{R}}_i$ plus the current cluster. For example at each step in Figure 1, $\boldsymbol{\mathcal{R}}_i$ is the set of green nodes (with black circles around), and $|\boldsymbol{\mathcal{R}}|_{max} = 4$, the maximum cardinality between all $\boldsymbol{\mathcal{R}}_i$ plus the current cluster (the node between the shaded areas). Typically the presence of transitive edges due to alternatives in T_a increases $|\boldsymbol{\mathcal{R}}|_{max}$ since from past clusters the analysis would reach configurations in future clusters (bypassing the current cluster) that needs to be kept in memory for future analysis steps. Therefore, the higher the $|\boldsymbol{\mathcal{R}}|_{max}$ metric, the higher the number of clusters (R_i) present in memory at once. This metric can be seen as a proxy for the efficacity of the Past-Free[ze] algorithm with respect to a given acyclic LTS, the smaller the $|\boldsymbol{\mathcal{R}}|_{max}$ the better Past-Free[ze] will perform. The best case is $|\boldsymbol{\mathcal{R}}|_{max} = 2$ that corresponds to a linear path (only two clusters need to be stored in memory regardless the cardinality of S_a). The worst case is $|\boldsymbol{\mathcal{R}}|_{max} = |S_a|$ that corresponds to the case where from the initial state all future states are reachable in only one step.

Freezability Metric This metric, $\boldsymbol{\mathcal{F}}$, captures the percentage of configurations that can be freed from memory with respect to the total number of configurations analyzed during reachability analysis.

$$\boldsymbol{\mathcal{F}} = \frac{|\boldsymbol{\mathcal{X}}|}{\sum_{i=1}^{|X_a|-1} |(-,i)|}, \tag{3}$$

where $|\boldsymbol{\mathcal{X}}|$ represents the total number of configurations reached either at end of the analysis (in which case $|\boldsymbol{\mathcal{X}}| = |\boldsymbol{\mathcal{S}}|$) or before failing due to combinatorial explosion (in which case $|\boldsymbol{\mathcal{X}}| < |\boldsymbol{\mathcal{S}}|$). $|X_a| - 1$ represents the number of clusters successfully freed from memory.

Relative-Progress Metric This metric represents the percentage of states in T_a already analyzed during the analysis. This offers the possibility to observe the progress of the analysis with respect to the acyclic part. It should be noted, however, that this metric shows only the relative progress and not the overall progress of the analysis which is equally dependent on the possible behaviors of the cyclic part.

$$\boldsymbol{\mathcal{RP}} = \frac{id}{|S_a|}, \tag{4}$$

where id is the variable id in Algorithm 1

DOI: 10.1002/stvr

3. PAST-FREE[ZE] IN PRACTICE: INTEGRATION WITH THE CONTEXT-AWARE VERIFICATION FRAMEWORK

To better characterize our technique, we integrated the Past-Free[ze] algorithm with the Context-aware Verification (CaV) approach. This methodological framework focuses on the isolation and the explicit specification of the environment conditions under which the SUS performs [18]. To this end, the CaV approach is supported by the OBP *Observation Engine* model-checking toolkit, which includes a language called CDL for specifying acyclic behaviors and exploits their characteristics to scale model-checking to large industrial systems.

After introducing the main motivations behind the CaV methodology, this section overviews the CDL language, and the OBP *Observation Engine* (which currently hosts an implementation of the PastFree[ze] algorithm).

3.1. The Context-aware Verification (CaV) Approach

To reduce the impact of the state-space explosion problem, in the case of large and complex systems, besides using techniques like partial-order reduction [41], the system designers manually tune the verification model to restrict its behaviors to the ones relevant to the specified requirements. This process is tedious and error prone since different versions of the model should be kept sound, in sync and maintained.

The *Context-aware Verification*(CaV) provides a structured approach for capturing the verification problem through a number of independent *verification contexts* (referred simply as contexts in the following), which explicitly represent the restricted model behaviors along with the requirements to be verified. The model is decomposed in two components: the system-under-study (SUS) and the environment. While the SUS specification is viewed as a black-box that never changes during the verification, the environment model is decomposed in multiple interaction scenarios, captured through the CDL formalism (Sec. 3.4.1). The verification contexts are created by associating to each interaction scenario the relevant properties that should be verified in each case. The verification process iteratively composes these contexts with the SUS to check the validity of the associated properties.

The CaV approach imposes a formal, methodical decomposition and classification of large requirements sets, a first step in overcoming the state-space explosion problem.

By limiting the scope of the system's behavior, the complexity of the verification is reduced; the decoupling of the SUS from its environment eases the maintenance of the system by allowing the environment refinement in isolation.

Furthermore, the possibility to analyze the SUS with a partial environment model gives valuable insights on particular context-dependent behaviors, enabling the designers to better focus their verification efforts.

Last but not least, the most important characteristic of the CaV methodology is the explicit identification of the acyclic behaviors, which allows to either automatically decompose the verification problems in a divide-and-conquer manner [18] (Section 3.2) and to apply the PastFree[ze] reachability algorithm (described in this study).

3.2. Recursive Decomposition of the Verification Problem

The acyclic nature of the interaction scenarios enables a simple yet powerful state-space decomposition technique, which relies on the automated recursive partitioning (splitting) of a given context in independent sub-contexts [18]. This technique, schematically presented in Figure 3, is systematically applied when a given reachability analysis fails due to the state-space explosion problem. After splitting a context, the sub-contexts are iteratively composed with the SUS, and the properties associated with the initial context are verified for all sub-contexts. Therefore, the global verification problem is effectively decomposed into smaller verification problems. Hence, verifying the properties on all these sub-problems is equivalent to verifying them on the initial case.

Figure 3 shows the recursive decomposition of an acyclic interaction scenario like the one presented in the Figure 1(a). In this case, if the composition fails, the interaction scenario can be split

Prepared using stvrauth.cls
DOI: 10.1002/stvr

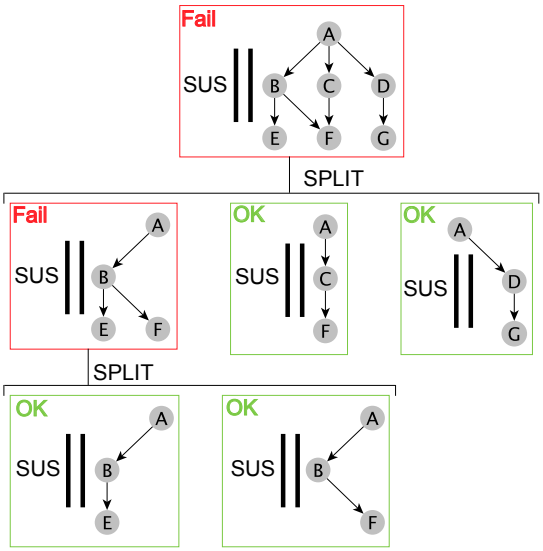


Figure 3. Recursive decomposition of the verification problem

into three independent scenarios, which should then be composed with the SUS. Amongst these, the first composition fails again and another split can be performed, which renders two new scenarios. Note that this decomposition can be performed until the initial interaction scenario is decomposed in linear paths and that this situation is the most advantageous for the Past-Free[ze] algorithm since only two clusters need to be present in memory at any step (the Reached-Future metric equals 2).

As opposed to the Past-Free[ze] algorithm that by exploiting the context acyclicity reduces the stress on memory during reachability, the use of this context partitioning technique reduces the memory requirements for a verification unit at the expense of having to run multiple explorations. It should be noted that, in the case of scenarios with a high degree of interleaving, the latter leads to multiple analysis of the same states (common prefix or suffix between the sub-contexts) which can slow down the verification process. However, in practice [8, 18, 38, 39], this tradeoff is worthwhile since this automated partitioning technique enables the analysis of large problems without the need to buy exponentially more memory. Moreover, the independent nature of the automatically generated sub-contexts, this problem can be partially addressed by distributing the verification over a network of computers.

3.3. Completeness and the Bounded Nature of the Analysis

Currently the CaV approach relies on the hypothesis that *it is possible to specify the sets of bounded environment behaviors in a complete way*. This implies mainly that the designer is able to identify the perimeter of the SUS and all possible interactions between it and its environment. Moreover, to satisfy the **acyclicity constraint**, all these interactions should be bounded. In other words, the state-space exploration performed corresponds to an unrolling of the system along a set of use-cases described in CDL, which are all of finite length.

The CaV methodology is generally not complete, in the sense that the unrolling of a system along a bounded interaction scenario potentially implies that some states remain undiscovered (e.g. the states unravelled by a longer scenario). This imposes virtually the same limitation as the bounded model-checking procedures [11]. Namely, that the analysis should be accompanied by a completeness proof showing that the bound b_{is} chosen for the interaction scenario enables the unrolling of its composition with the system to a depth at least equal to the **Completeness Threshold** $\mathbb{C}$. Moreover, given a cyclic environment and an arbitrary system, $\mathbb{C}$ is an upper bound on b_{is}. Hence, if the Completeness Threshold of the composition is known it is sufficient, but not necessary, to unroll the cyclic environment model to that depth to achieve completeness.

DOI: 10.1002/stvr

For the verification of safety properties the completeness threshold is given by the reachability diameter r_d (the minimum number of steps required for reaching all reachable states) [31].

In the context of CaV we are currently investigating: *a)* the integration of a bounded unrolling of cyclic environment models, which will generalize the approach to arbitrary systems; *b)* the possibility to automatically compute the minimal b_{is} that guarantees that the composition of the interaction scenario with the SUS reaches the Completeness Threshold, which provides the necessary conditions for the completeness proof.

3.4. Context Specification with CDL

To specify the environment conditions separately from the SUS (System-Under-Study), the CDL language was introduced as part of the OBP toolkit [17]. The core concept of the CDL language is the **context**, which associates the requirements to be verified to an acyclic LTS communicating asynchronously with the system. The interaction of the SUS with the environment is specified through a number of interaction scenarios. The interleaving of these interaction scenarios generates an LTS representing all the behaviors of the environment, which can be fed as input to model-checkers (currently SPIN & OBP). Moreover, CDL enables the specification of requirements about the system's behavior as properties that are verified by the OBP *Observation Engine*. These properties expressed through property-pattern definitions [20] are based on events (e.g. variable x changed) and predicates.

The following sections overview the CDL language to specify these interaction scenarios (Sec. 3.4.1), and to specify properties for capturing the requirements (Sec. 3.4.2), and the top-level structure of a verification context (3.4.3). The reader should refer to [17] for an in-depth presentation of the CDL language semantics.

3.4.1. Environment Modeling Through Interaction Scenarios To express the environment interactions with the SUS, the CDL language is based on the Message Sequence Charts (MSCs) [42] which grammar is as follows:

$$\begin{array}{lll} C & ::= & M \mid C_1; C_2 \mid C_1 [] C_2 \mid C_1 \| C_2 \\ M & ::= & 0 \mid a!; M \mid a?; M \end{array}$$

A CDL environment is described as a finite generalized Message Sequence Chart (MSC) (C) which is either (1) a sequence of event emissions $a!$ and event receptions $a?$, or (2) a sequential composition (seq denoted ;) of two MSCs ($C_1; C_2$), or (3) a non-deterministic alternative (alt denoted []¶) between two MSCs ($C_1 [] C_2$), or (4) a parallel composition (par denoted $\|$) between two MSCs ($C_1 \| C_2$). An emission event is an asynchronous communication from the environment to the SUS. Similarly, a reception event is an asynchronous communication from the SUS to the environment.

The CDL language uses the concept of events to describe such emission/reception events between the SUS and the environment, as well as to describe internal changes within the SUS (e.g. a variable change, or events sent between the subparts of the SUS). For example, the emission of an event named E to a process P that represents part of the behavior of the SUS in the form of an automaton, is described as: `event Handle is { send E to {P}1}`, where `{P}n` refers to the n^{th} instance of the process P).

The composition of MSCs (C) is introduced by the CDL *activity* keyword. For instance, the interleaving of two `E` events with one failure event can be expressed through the following activities:
`activity 2E is { loop 2 E }`
`activity EAndFailure is { 2E || injectFailure }`, where `injectFailure` is an emission event. The `loop n` construct is only syntactic sugar for expressing long chains of sequential interactions.

¶The alt operator was denoted + in the original syntax, in this study we have used [] to match the actual CDL grammar.

DOI: 10.1002/stvr

3.4.2. Property Specification The CDL language provides three constructs for expressing safety and bounded-liveness properties: *a)* predicates, for expressing invariants over states; *b)* observers, for expressing invariants over execution traces; *c)* property patterns, for simplifying the expression of complex properties.

The predicates, defined by the *predicate* keyword, are basic propositional logic formulas; they can either be used as assertions in a context, or in the definition of observers or property patterns. For example, `predicate pRed is { {proc}1:red_light=true }` expresses that the variable `red_light` of {`proc`}1 is true.

The observers are timed automata used to express invariants over execution traces. Besides being deterministic and complete, their particularity is that they are composed synchronously with the system, and advance by observing the occurrence of events, like the changes in the valuation of a predicate. A property encoded by an observer is violated if, during the execution of the system, the observer reaches a predefined rejection state (*reject*). For example a deadline property, such as: "*after p1, p2 should occur before 10 time units*", translates to the three-state CDL observer presented in Listing 1. Initially in the **start** state, this observer moves to **s1** if the predicate p1 is true (initializing the clock ck). If the p2 predicate is or becomes true before 10 time units it returns to the **start** state, if not it reaches the **reject** state (also disabling its clock). If the **reject** state is reached then p2 did not occur before 10 time units.

Listing 1: CDL-based property specification

```
property oR2 is {
        clock ck;
        start -- p1 / ck := 0 -> s1;
        s1 -- p2 / ck := -1 -> start;
        s1 -- ck >= 10 / ck := -1 -> reject}
```

The predicates, and observers are simple yet powerful mechanisms for property specification. However, for complex properties they tend to become hard to understand and manipulate, mainly due to the large number of subtle interdependencies between the events manipulated, their occurrences and scope. To address this issue the CDL language offers support for property-pattern specification, inspired by the pattern language introduced by Dwyer [23]. The interested reader should refer to [16] for more details on this aspect, which is beyond the scope of this paper.

3.4.3. The Toplevel of a Verification Context The CDL contexts provide well defined verification units that associate interaction-based scenarios with relevant properties. A context, introduced through the cdl keyword, is structured in two sections: the *property assertion* part, and the *scenario specification* part. The *property assertion* part contains the list of predicates that should be globally satisfied by the system (i.e. true in all states of its state space), and the list of observers that should be composed with the system. The *scenario specification* part defines the interactions of the environment with the SUS and is itself decomposed into an initialisation sequence (introduced by the $init$ keyword) and the core scenario (introduced by the $main$ keyword).

Listing 2: The structure of a CDL verification context

```
cdl context1 is {
        assert predicate1, predicate2
        properties observer1, observer2
        init is { initialization sequence }
        main is { interaction scenario }
}
```

Listing 2 shows the structure of a typical context. Both the *init* and the *main* blocks can be empty. Doing so and specifying the environment within the same formalism as the SUS instead of CDL, leads to the traditional model-checking setup.

DOI: 10.1002/stvr

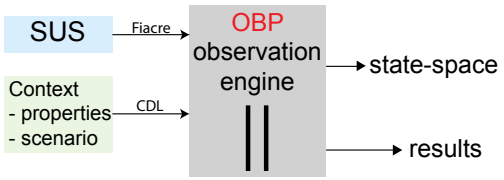


Figure 4. Context-aware Verification with OBP *Observation Engine*

3.5. OBP: A Context-aware Verification Toolkit

The practical use of the CaV approach is enabled through the OBP toolkit, which provides built-in support for the CDL language and its composition with the SUS models. At the core of the toolkit lies the OBP *Observation Engine* providing the verification kernel.

Figure 4 shows a global overview of the OBP *Observation Engine*. The environment is decomposed in verification contexts expressed with the CDL language. The SUS is described using the Fiacre modeling language [25].

Fiacre is a textual language similar to Promela yet not deeply bound to a particular verification toolset. The language supports behavior coordination through either shared variable communication or synchronous channels (which can be seen as an extension of the LTS label synchronization). A Fiacre models is structurally organized in Processes (primitive units of behavior - equivalent to an automaton) and Components (hierarchical instantiation units, built from the parallel composition of process instances or other component instances). A prominent feature of Fiacre is the possibility to express timing constraints (in the style timed automata).

The OBP *Observation Engine* verifies the given set of properties with a reachability strategy on the graph induced by the parallel composition of the SUS with the interaction scenario specified in the context. During the exploration the *Observation Engine* captures the occurrences of events and evaluates the predicates after the atomic-execution of each transition. It then updates the invariants and the status of all observers involved in the run. A report is generated, at the end of the exploration, showing the valuation of all invariants and the status of the attached observers. Moreover, the resulting LTS can be queried to find either the system states invalidating a given invariant or to generate a counter-example based on the $reject$ state of a given observer.

To foster the generality of CaV and its complementarity with existing model-checkers the toolkit provides a bridge [17] to the TINA [6]. This shows the possibility of benefiting from the methodological advantages of the CaV approach in conjunction with off-the-shelf verification tools.

The OBP toolkit performance is comparable to existing model-checking tools as assessed by a study that we conducted using the BEEM benchmarks [34]. This study compared OBP with the SPIN and DIVINE model checkers. Specifically, OBP was able to explore most of the 167 test cases of the BEEM benchmarks, as well as to complete some of the test cases that could not be validated by SPIN nor DIVINE because of the state-space explosion of these test cases. For the successfully explored cases, OBP obtained the same number of states/transitions as DIVINE. Globally, OBP is on average 2.25 times faster than DIVINE (with a standard deviation of 1.54). SPIN is usually faster than OBP, however, OBP was able to explore more test cases then SPIN without reaching state explosion (133 complete explored test cases for OBP, 138 for DIVINE, and 110 for SPIN[‖]). The comparative table of the BEEM benchmarks is available on the OBP website[**].

Analysis of timed automata: The OBP toolkit enables also the analysis of timed-automata with performances comparable with the TINA toolkit [19]. Note that, in the case of systems modeled with timed automata, each configuration of the system is extended to include a symbolic representation

[‖] Note: the BEEM benchmark lacks the SPIN/Promela version of 31 test cases
[**] http://www.obpcdl.org

DOI: 10.1002/stvr

Table I. Overview of the reachability results

	OBP	Past-Free[ze]	Gain
passed tests	38%	56%	1.47X
Cumulated results for all test cases			
states	62 845	145 399	2.31X
transitions	85 641	194 471	2.27X
time reach.+ IO (sec.)	9283	15582	0.59X
Cumulated time for the tests that passed both with OBP and Past-Free[ze]			
time (sec.)	112	85	1.31X
time +IO (sec.)	112	930	0.12X

of time encoded through a Difference-Bound Matrix (DBM) [5]. In this case the LTS manipulated by the reachability algorithm is a symbolic representation of the system, where time is represented through equivalence classes, named zones, and encoded with DBMs. Through this construction the reachability checking in the case of timed automata (which are infinite-state automata) is equivalent to the reachability checking in the finite automata with zones.

The integration of PastFree[ze] in the OBP *Observation Engine* was realized by the direct implementation of the algorithm in Java and by coupling it to the verification core. It should be noted that since the PastFree[ze] approach relies on the LTS model, it could be integrated in other model checkers, such as SPIN [29].

4. PAST-FREE[ZE] EVALUATION: SYNTHETIC BENCHMARK

As a preliminary assessment of the Past-Free[ze] algorithm, we defined and analyzed a benchmark of 50 acyclic CDL contexts connected with a generic cyclic system (encoded in Promela [29] and in Fiacre [25]).

In this section, we first briefly overview the results of the experiment, then we describe the construction of the benchmark using the OBP *Observation Engine* toolkit. The reachability results of the Past-Free[ze] technique are discussed by (1) comparing it to the state-space exploration strategies implemented in SPIN and OBP, and (2) analyzing the impact of Past-Free[ze] on the analysis time. Moreover, the positive results presented in this section are reinforced, in Sec. 5, by the promising results obtained on two realistic case studies from the automotive and aerospace industry.

4.1. Summary of the results

Table I overviews the results obtained using the Past-Free[ze] algorithm and compares them with those obtained using OBP *Observation Engine*. The passed tests row shows the percentage of test cases that were successfully explored (no state-space explosion), as well as the gain of our technique. The middle part of Table I shows the cumulated results, in terms of LTS size and run-time, of the exploration of 50 test cases. The lowest part of Table I shows the cumulated run-time for the analysis of all the test cases that passed with both BFS and Past-Free[ze] algorithms. Overall the Past-Free[ze] algorithm enables the exploration of larger state-spaces than OBP (2.31 times larger for the benchmark considered in this study).

4.2. Evaluation Setup and Material

The results presented in this section were obtained on a 64 bit Mac OS computer running OS X 10.9.1, with a 2.53 GHz Intel Core i5 processor, and 8GB RAM memory. For SPIN we used the version 6.4.0. The OBP *Observation Engine* distribution version 1.4.8 was used, which includes an implementation of the Past-Free[ze] algorithm. For practical reasons the size of the heap allocated

Prepared using stvrauth.cls
DOI: 10.1002/stvr

to the OBP *Observation Engine* was limited to 2GB. In explicitly identified cases the memory limit was increased to 3GB.

The raw results presented in this study along with the source files and an OBP *Observation Engine* distribution including the Past-Free[ze] algorithm are available for download on the OBP *Observation Engine* website at `http://www.obpcdl.org`.

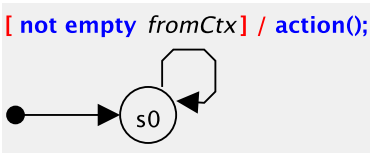
The traditional state-space exploration strategies (implemented in SPIN & OBP) are based on depth-first search or breadth-first search. Both these techniques unravel the same states and keep them in memory leading to explosion. Similarly, the Past-Free[ze] algorithm discovers the same state-space, however by freeing parts of it from memory it pushes the state-space explosion limit further. Furthermore, Past-Free[ze] is independent of the techniques used to store the tables of visited configurations (i.e. hashmap, bloom filter, etc.).

4.3. Benchmark Synthesis

In order to evaluate the DAG-directed reachability analysis technique on a wide range of acyclic behaviors, we created a synthetic benchmark of 50 acyclic contexts connected with a generic cyclic system.

Listing 3: SUS description with Fiacre (cyclic) and the corresponding automaton representation

```
1  type StateVector is array log2NStates of bool
2  type d3 is array 1024x1024x1 of int
3  type ContextQueue is queue 1 of int
4
5  process Behavior (&fromCtx : ContextQueue) is
6  states s0
7  var sV : StateVector, data : d3
8  from s0
9    on not empty fromCtx //transition guard
10   sV[first fromCtx] := not sV[first fromCtx];
11   data[0][0][0] := (data[0][0][0]+1) % NStates;
12   fromCtx := dequeue fromCtx;
13   to s0
14
15 component sys is
16 var      Behavior_1 : ContextQueue := {||},
17          toContext  : ContextQueue := {||}
18 par      Behavior(&Behavior_1)    end
19 sys
```



Cyclic System-Under-Study The cyclic part of our system, presented in Listing 3, is implemented using the Fiacre language. It consists of a single *Behavior* process (instantiated once – line 18 in Listing 3) that receives orders ($nEvents$ different orders) from the environment (modeled as a context). This process has only one transition looping in $s0$. This transition is enabled when an order is available on the $fromCtx$ queue, and it is blocked otherwise ($empty\ fromCtx$). According to the received order, the *Behavior* process flips a bit in its sV array, hence the system has an exponential number of configurations (2^n). Moreover, the *Behavior* owns a 3 dimensional integer array which enables us to increase the memory pressure (an 1024 by 1024 integer matrix takes at least 4MB of memory). As a result, a system receiving 10 orders (interleaved) will have 2^{10} configurations of 4MB each, with a total size of 4GB of memory for storing all the reachable states.

In our case, the number of reachable configurations is larger due to the asynchronous (FIFO-based) connection between the context and the system. As a consequence, when considering the 10 interleaved events we obtain an LTS with 6 144 configurations and 10 240 transitions, which sums up for over 24GB of required memory. However, in practice the LTS is only

DOI: 10.1002/stvr

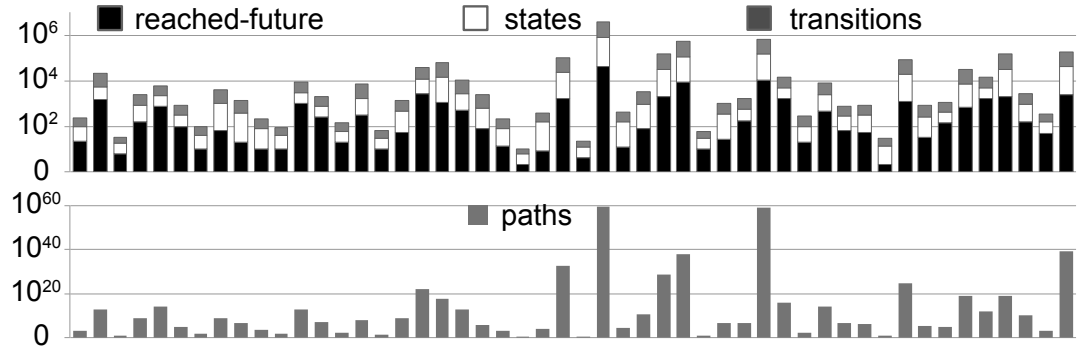
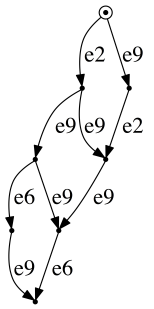


Figure 5. 50 CDL benchmark characteristics

avg. 13GB††, due to the configuration compaction strategy used in OBP *Observation Engine* (similar to the *-Dcollapse* option in SPIN) that enables part sharing between configurations.

Listing 4: Acyclic context description using the CDL language and LTS representation of cdl c_{26}.

```
1  event e_i^{1..10} is {
2          send i to {Behavior}1 }
3
4  activity seq_8 is {
5          event e_2; event e_9; event e_6 }
6
7  activity alt_1 is {
8          seq_1 [] seq_2 }
9
10 cdl c_26 is {
11         main is { seq_8 || event e_9 } }
```



Acyclic Context Generation The *Behavior* process is asynchronously composed with 50 different acyclic contexts, each of which models different event patterns that can be sent to the system. The purpose of the acyclic context generation strategy, presented in this section, is the generation of multiple independent scenarios with different shapes (in terms of the graph structure), which are used to objectivelly evaluate the Past-Free[ze] algorithm. These contexts are expressed using the CDL language (see Listing 4 for an example) and are randomly generated using a configurable generator. The context generation strategy as defined by Algorithm 2, iteratively builds the contexts from primitive CDL constructs. It first generates $nEvents$ (line 1,2 in Listing 4), then it randomly chooses $maxSeq$ events from P and builds $nSeq$ sequences in P. $maxAlt$ elements from P are randomly chosen to further create $nAlt$ alternatives, which again are added in P. To control the complexity of the sequences and alternatives the previous two steps are repeated N times. The generation ends by building the $nPar$ (**cdl**) contexts by parallely composing $maxPar$ elements from P.

For the benchmark presented in this study we used the following parameter set:

$$N = 2,\ nEvents = 10,\ nSeq = 10,\ nAlt = 10,\ nPar = 50,$$

which generated 50 CDL contexts using 20 alternatives, 19 sequences and 10 event types.

Figure 5 presents the characteristics of the generated sample. The topmost diagram shows the number of states (transitions) as well as the *reached-future metric* (cf. Sec. 2.4). The *reached-future metric* (black bars) is always smaller or equal to the number of states (black + white bars) due to the

††This reachability analysis was performed using the Past-Free[ze] algorithm on OBP *Observation Engine* setting the limit of usable heap at 3GB

DOI: 10.1002/stvr

Algorithm 2 Random CDL Generation Algorithm

```
constant maxSeq = 10, maxAlt = 5, maxPar = 3
function CONTEXTGEN(N, nEvents, nSeq, nAlt, nPar)
    set of primitives P ← ∅
    P ← P ∪ genEvents(nEvents)
    for i ∈ [1..N] do
        P ← (P ∪ genSeqs(nSeq, P)) ∪ genAlts(nAlt, P)
    end for
    return contexts ← genParallel(nPar, P)
end function
```

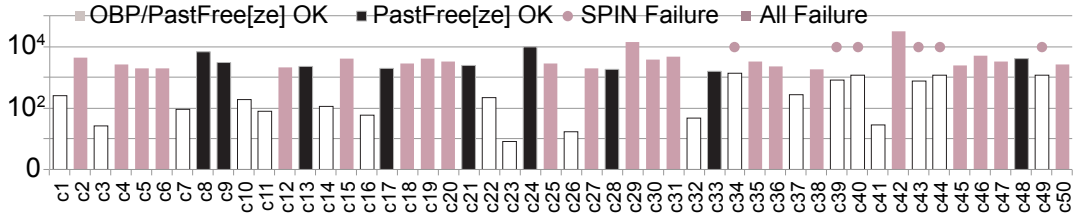


Figure 6. Number of explored states and state-explosion status.

DAG structure. The number of transitions (black + white + gray bars) has a lower bound induced by the spanning tree of the graph (n-1 transitions for n states) since we do not consider disconnected graphs. In this sample, the number of states (transitions) varies between 4 (4) up to 709 776 (2 955 036) with an average of 22 908 (90 170). The lower diagram shows the total number of paths in the generated acyclic DAGs, which varies between 2 for $c22$ to more than 10^{59}.

The *reached-future metric*, in Figure 5, gives the maximum number of sets R_i that will be used at once during the Past-Free[ze] exploration. In the most favorable case, for a single path (linear) context ($|\boldsymbol{\mathcal{R}}|_{max}$ equal to 2), at any given time at most two hash-tables will be used (the one corresponding to the current context state, and the one for the next one). For the sample in Figure 5 the $|\boldsymbol{\mathcal{R}}|_{max}$ averages 1620 sets.

4.4. Reachability Results

The results of the reachability analysis of the benchmark with both OBP & SPIN and Past-Free[ze] approach are presented in Figure 6, which shows the number of explored states and the state-explosion status. Among the 50 test cases 16 were successfully analyzed by SPIN (the white bars without red dot) [‡‡]. OBP finished successfully for 19 cases (the white bars). The Past-Free[ze] algorithm clearly improved on this baseline with 9 more cases that have passed successfully (the black bars), thus enabling the full analysis of systems with over 7 times more states than the DFS/BFS reachability. In absolute values the largest case successfully explored using SPIN or OBP was $c34$ (analyzed successfully by OBP) with 1356 states while the Past-Free[ze] algorithm reached 9628 states for the $c24$ case.

Though the state-space explosion is still present (22 test cases failed, cf. All Failure bars – pink bars – in Figure 6), even in this cases the Past-Free[ze] algorithm performed better analyzing up to 14 times more states before state-space explosion than the other two. Even if the analysis is not exhaustive, in this case, pushing the analysis limits can however enable the early detection of property violation.

[‡‡]In the case of SPIN some of the failed cases (that have failed with both OBP and Past-Free[ze]) –pink bars in Figure 6– posed some problems during the Promela compilation process, in these cases the analysis was either stopped after 2 hours of compilation (c27, c31, c35, c42, c47, c50) or abruptly crashed with a segmentation fault (c25, c30)

DOI: 10.1002/stvr

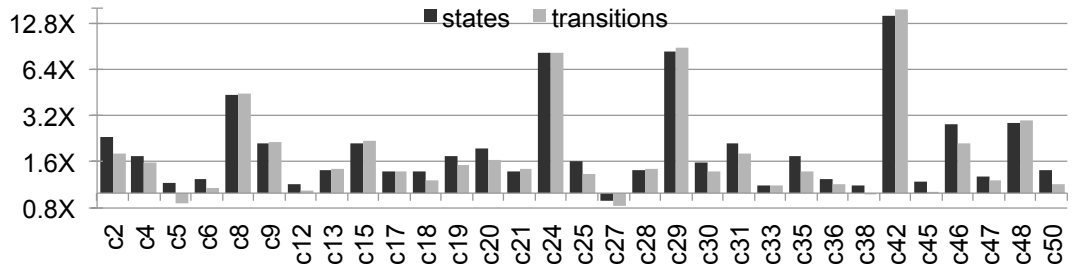


Figure 7. Exploration gain (X)

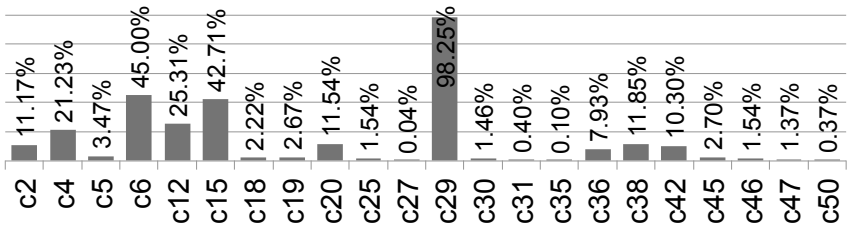


Figure 8. Percentage of the explored context states for state-explosion cases

Reachable-state gain Figure 7 shows the gain in terms of number of reachable states (PF_{states}/BFS_{states}) and transitions ($PF_{trans.}/BFS_{trans.}$) analyzed by Past-Free[ze] over BFS for the state-explosion cases. In total the Past-Free[ze] algorithm explored 2.31X more states and 2.27X more transitions, which correspond to the exploration of 82554 more states and 108830 more transitions than the BFS baseline. For some cases ($c5$, $c27$ and $c38$ in Figure 7) negative gain values were obtained, which seem to contradict the previous statement. Indeed in these state-explosion cases our approach explored less states (for $c27$) and/or transitions (all three) than the BFS baseline. The reason behind this hides behind the different exploration order of the reached states: the BFS uses the natural (arbitrary) order in which the states were reached while the Past-Free[ze] orders the reached states according to the partial ordering of the acyclic context. The $c27$ case failed before reaching the number of states discovered by BFS, showing a particularly unfavorable case where the data-structure overhead of Past-Free[ze] along with the ordering strategy hinders the reachability analysis. Nevertheless the positive gain for 19 of the 22 state-explosion cases outweighs the occasional negative ones, especially since these cases appear exclusively in the case of state-space explosion.

Relative progress As stated in Sec. 2.4, a secondary advantage of the DAG-directed exploration strategy is the possibility to observe the relative progress of the reachability analysis with respect to the linear order of the acyclic part. Figure 8 shows such results for the 22 state-explosion cases of the benchmark. While these results remain subject to the varying complexity of the cyclic part with respect to any given DAG state, the designer (who knows his system) can benefit from these quantitative measurements. For example, considering our benchmark we can infer that the $c29$ case (98.25%) is closer to the finish line than the $c27$ (0.04%) – the largest considered context. Hence by using a platform with a little more memory we can hope to successfully finish this case – a decision that cannot be taken objectively in the BFS case. By increasing the memory limit to 3GB (from the 2GB initially used) the test case ($c29$) passes reaching 15 309 states and 23 331 transitions (1586(2935) more states(transitions) than before).

Freezability rate Finally, to further highlight the potential gain of the DAG-directed approach to reachability analysis, Figure 9 shows the *freezability rate*, $\mathcal{F}$, the percentage of the reached states (passed and "explosion" cases) that can be freed from the memory during the execution. On the

DOI: 10.1002/stvr

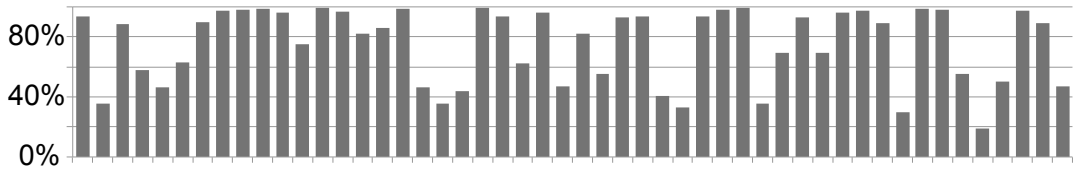


Figure 9. *Freezability rate*: Percentage of reachable states that can be freed during reachability analysis

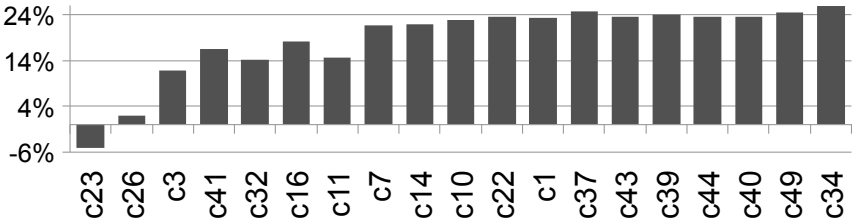


Figure 10. Percentage improvement of time due to less collisions

studied benchmark we obtained an average ratio of 75% with the worst case of 18.97% for $c46$ and the best case of 99.63% for $c34$.

4.5. Impact on the Analysis Time within OBP

To analyze the time performances of the Past-Free[ze] algorithm relative to OBP, the 19 test cases that passed were run 10 times for each algorithm and the average results were reported. This choice was made mainly since, in the state-space explosion cases, the overhead of the garbage collector dominates the run-time (as mentioned before the OBP *Observation Engine* exploration engine is implemented in Java). With a better object-oriented encoding of the state space we could solve this issue by manually handling the memory management for implementing the past state freeing strategy. However, these optimizations are out of the scope of this paper and will be addressed in future versions of the OBP *Observation Engine*.

Figure 10 shows the speed improvement of our algorithm over OBP, in the case where the past states from R_i are not saved to disk. The results are sorted in an increasing order based on the number of reachable configurations. The observed gain comes mainly from the decomposition of the global hash-table into $|S_a|$ independent hash-tables, which reduces the number of hash collisions. In the BFS case the number of hash-collisions increases with the number of explored configurations. In our case the hash-collisions are inversely proportional to the number of context states of the system. In consequence, the larger the context the faster the Past-Free[ze] algorithm performs compared to BFS. The negative value for the $c23$ case and the small value (under 4%) for $c26$ can be explained by small context size in these cases (4 and 6 states respectively). Moreover, since the total execution time of these two cases is under 2.5 seconds the interferences from the execution environment might have introduced further noise.

However, even though just by using the hash-table decomposition Past-Free[ze] algorithm could analyze on average 80 $states/sec.$ (with a max on the cases presented in Figure 10 at almost 100 $states/sec.$) compared to 63 $states/sec.$ on average for BFS, a tradeoff has been made in order to reduce the memory requirements during the reachability analysis by dropping the past configurations and saving them to disk. In this case the time performances of the Past-Free[ze] algorithm become clearly IO bounded when the past configurations are saved. Figure 11 shows the average overhead (over the 10 runs) for the 19 measured test cases, which reaches a factor of $7.68X$ (in average) compared to BFS. To address this issue, in the future we plan to perform IO optimizations.

DOI: 10.1002/stvr

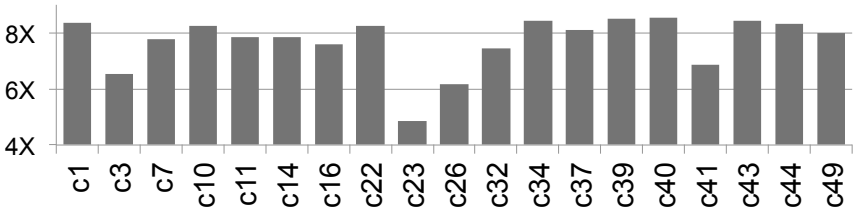


Figure 11. Time overhead due to IO operations

Figure 12. Cruise Control System analysis results.

5. PASTFREE[ZE] EVALUATION: REALISTIC CASE-STUDIES

In this section, we empirically show the effectiveness of our approach on two realistic case studies: an automotive Cruise Control System (CCS) and an aeronautics Landing Gear System (LGS). We then discuss some of the advantages offered by the PastFree[ze] technique for the analysis of the LGS system.

5.1. Evaluation Setup

For these case studies, the results were obtained using OBP v.1.4.8, which ran on two 64-bit Linux configurations, referred to as L64 and L128, with respectively 64GB and 128 GB of RAM. Both case studies have been defined using the CaV methodology with the Fiacre and the CDL languages. The Cruise-Control System has 1553LOC and the Landing Gear System has 3000LOC and the source code is available at `http://www.obpcdl.org`.

Realistic Case-Study: Cruise Control System The effectiveness of our approach is shown on a realistic case-study from the automotive domain, which is fully described in [39]. To this end, the CaV methodology was used for modeling and requirement validation of a Cruise-Control System, a system that automatically controls the speed of cars. Using this approach we verified three important requirements of the CCS, identifying one subtle concurrency bug that could lead to very dangerous situations. Furthermore, the importance of the CaV approach is emphasized through the successful analysis of up to 4.78 larger state-space than traditional BFS. Result which was made possible by the complementarity of the PastFree[ze] algorithm with the recursive state-space decomposition strategy [18], pioneered in OBP *Observation Engine*. Some quantitative results from this study are summarized in Figure 12 emphasising the advantages of the PastFree[ze] algorithm. It should be noted that these results show the net gain (4.78X), the actual number of states effectively explored in the third case being 1.92 times larger (779 739 813 states) since the recursive decomposition strategy does not produce disjoint contexts. Hence, in reality the combination of the two techniques enabled the exploration of a ”gross” state-space 9.18 time larger than what could have been achieved using only BFS or DFS reachability.

Note that the analysis overhead, arising due to the disjoint contexts, could be effectively reduced, in practice, by reusing the intermediate analysis result between contexts that share a common prefix or suffix, however this optimization is not yet implemented.

DOI: 10.1002/stvr

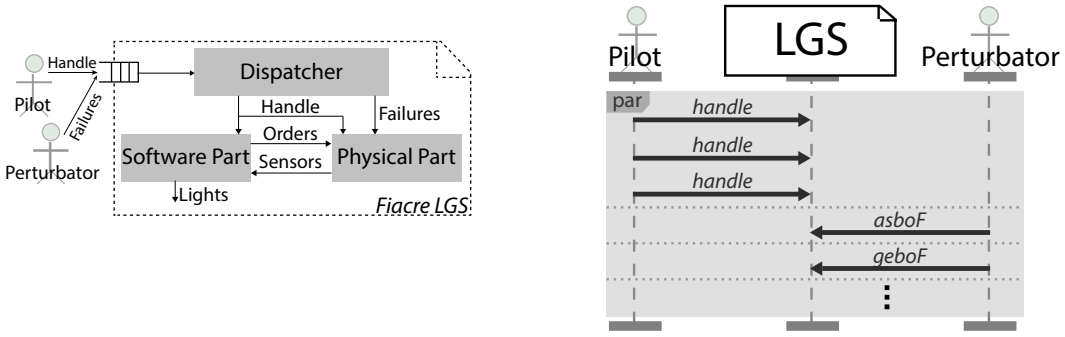


Figure 13. LGS SUS components (left) and an LGS environment interaction scenario (right)

Table II. Fiacre processes for the Physical Part (left) and Software Part (right)

Process name	# of states	# of instances	Process name	# of states	# of instances
Analog Switch	18	1	Door sensor synth.	8	3
General_EV	34	1	Gear sensor synth.	8	3
Generic_EV	24	4	EV Manager	52	1
Gear	23	3	Status Manager	10	1
Door	20	3			

Landing Gear System To answer a challenging case-study proposed by the ABZ'2014 conference [1], we applied the CaV methodology for modeling an Aircraft Landing Gear System [9,38]. A LGS is in charge of maneuvering the landing gears and the associated doors of an airplane, under the control of a pilot. The goal was to verify the software monitoring system in the presence of failures. We examine the results of the LGS experiments in the next section.

5.2. PastFree[ze] for Timed Systems Reachability: The Landing Gear System Case

In this section we apply our context-aware verification approach to the LGS case study, we overview the LGS modeling using the Fiacre language and the environment specification using CDL. We then examine the reachability analysis results of this model using PastFree[ze] with regard to the presence of multiple failures 5.3 and to arbitrarily long contexts 5.4.

5.2.1. Overview of the LGS model The LGS model, presented in Figure 13 (left), is composed of two parts: a model of the software part, and a model of the physical part, communicating through urgent signals. The environment of the LGS is composed of two agents: the pilot sending *handle* events to change the handle position (from down to up and vice-versa), and a virtual agent called *Perturbator* injecting *failure* events in the physical components (Figure 13, right part). The interactions from the environment (i.e., handle and failure events) to the LGS model are managed by a specific component called *Dispatcher*. Inputs are received through a FIFO channel and are dispatched immediately to the software part (*handle*) and to each physical component (*failures*). Outputs (i.e., the lights status) are modeled through global variables set by the software part.

The physical part is the parallel composition of 12 instances of the following Fiacre processes: *a*) *Analog Switch*, implementing the behavior of the analog switch; *b*) *General_EV*, implementing the behavior of the general electro-valve; *c*) a generic process *Generic_EV*, implementing the behavior of one electro-valve; *d*) a generic process *Gear*, implementing the behavior of one gear; *e*) a generic process *Door*, implementing the behavior of one door. Table II(left) shows the number of states of each of these processes along with the number of times each one is instantiated in the model.

Prepared using stvrauth.cls
DOI: 10.1002/stvr

Similarly, the software part (overviewed in Table II(right)) is the parallel composition of 8 instances of the following Fiacre processes: *a*) a generic process *Door sensor synthesis*, which computes the door state (*closed*, *open*, or *intermediate*) from the values returned by the sensors; *b*) a generic process *Gear sensor synthesis*, which computes the gear state (*retracted*, *extended*, or *intermediate*) from the values returned by the sensors; *c*) *EV Manager*, which executes the extension and retraction sequences according to the handle position and the values returned by the sensors; *d*) *Status Manager*, which computes the status (on or off) of the three lights in the cockpit.

5.2.2. Modeling the Context The environment of the LGS is modeled with the CDL language. It is composed of the interleaved actions of two context actors, namely the pilot sending up/down commands through its handle, and a virtual actor (named Perturbator) introducing failures into the system. The pilot behavior is represented through a CDL activity composed of a sequence of N $handle$ events sent to the $Dispatcher$ process (see the first two lines of Listing. 5).

The $Perturbator$ actor encodes all the possible failure configurations composed of sequences of 1 up to 3 failures taken from the total of 18 failures that have been identified. It should be noted that between the first 12 failures there are groups of 2 exclusive failures (ex. the analog-switch cannot be blocked in the opened and closed state at the same time). Taking these exclusion rules into account it follows that there are 885 possible failure configurations as follows: *a*) 18 possible configurations with 1 failure. *b*) 147 possible configurations with 2 failures (and 6 excluded failures). *c*) 720 possible configurations with 3 failures (and 96 excluded failures). Each of these failure scenarios are encoded as a CDL activity (Listing 5 lines 7–14), named FailureContext^x_k, where $x \in [1 \dots 3]$ is the number of failures and k is the id of a given configuration from the set of the ones possible with x failures (ex. $k \in [1 \dots 147], for\ x = 2$). The $Perturbator$ actor is then represented as a CDL activity that non-deterministically chooses one of these failure configurations to play, see lines 11–14 in Listing 5.

Listing 5: Overview of the CDL environment description.

```
1  event Handle is  { send HANDLE to {Dispatcher}1}
2  activity PILOT is { loop N event Handle }
3
4  // analog-switch blockedOpen Failure
5  event asboF is { send ASBO_FAILURE to {Dispatcher}1}
6  activity FailureContext^1_k is { event k^th failure }
7  activity FailureContext^2..3_k is {
8  ...// all permutations of k^th 2(or 3) failures}
9  activity Perturbator is {
10     FailureContext^1_1 [] ... [] FailureContext^1_18
11   [] FailureContext^2_1 [] ... [] FailureContext^2_147
12   [] FailureContext^3_1 [] ... [] FailureContext^3_720}
13
14 cdl scenario_885_failure_configurations is {
15     properties oR_1, ..., oR_n
16 // environment model
17     init is { act_init }
18     main is { PILOT || Perturbator }
19 }
```

The CDL specification of the global environment, Listing 5 lines 17–18, consists of the initialization of the SUS (line 17) followed by the asynchronous interleaving of the $Pilot$ events with the $Perturbator$ failure sequences. Note also the association of the properties to be verified in this context (line 15). The presentation of these properties is, however, out of the scope of the current study, the interested reader can refer to [38] for details. To handle the analysis of the large state-space induced by this context, it was decomposed in 885 subcontexts, each one interleaving the PILOT actor with the respective failure configuration.

DOI: 10.1002/stvr

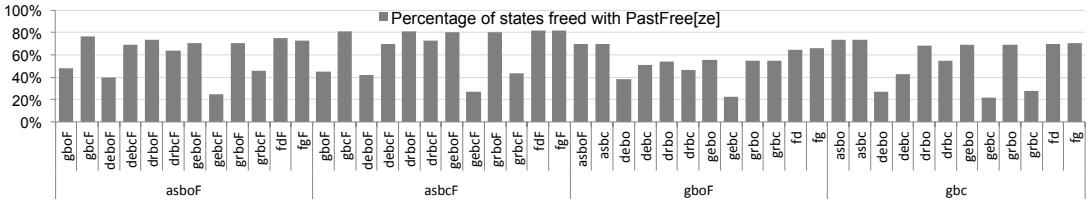


Figure 14. The percentage of the state-space freed from memory using the PastFree[ze] reachability for 48 two-failure CDL contexts.

5.3. Reachability Analysis with Multiple LGS *Failures*

In this section, we report the reachability results of the LGS on a subset of 48 verification contexts with two failures injected. Compared to Berthomieu et al.'s experiment [7] that uses a similar model of LGS, we were able to analyze the model with more than one failure without reaching state-space explosion (cf. Figure 14).

The analysis of most contexts finishes successfully, though some failure combinations unravel very large state-spaces. For instance the analysis of three pilot interactions interleaved with the occurrence of a general electro-valve blocked in the open position failure (gboF) followed by a gear extension electro-valve blocked in a closed position (gebcF) failed on $L128$ after unravelling 162 780 101 states.

Using the PastFree[ze] reachability algorithm, we enabled the exploration of a larger state-space. The explosion limit line on $L64$ was pushed from 35 701 272 states to 69 553 139 states (in Figure 14), representing the exploration of a state-space almost twice larger (1.94 times larger).

It should be noted that the LGS model is encoded using timed-automata with a total of 14 clocks. The analysis of this model relies on a dense representation of time, through DBMs (as described in Sec. 3). A direct consequence of this representation is that each state contains a 14 by 14 DBM encoding the clock relations (the size of this matrix is 392 bytes=14*14*2 bytes). Thus, from the 64GB of memory available on $L64$ over 14GB are used only for the storage of the 35 701 272 DBMs. If it were to store all 69 553 139 states in memory, over 27GB would have been dedicated only to the storage of DBMs.

Figure 14 shows the percentage of the state-space that was freed from memory during the analysis of each of the 48 verification contexts. This ratio varies between 20% and 80%, with an average around 49%. Almost 1 billion (981 437 225) of the 2 billion states analysed were freed from memory during the analysis.

5.4. Arbitrarily-Long Finite Contexts

While our PastFree[ze] technique made it possible to explore some cases of interleavings of two/three failures without state-space explosion, with an infinite number of pilot interactions the state-space explosion was systematically observed. To cope with this issue, we could restrict the number of pilot interactions, as was done in the previous section. Hence the Pilot actor becomes acyclic (sends a given number of handles then stops), and can be integrated in the CDL context specifications. However, in this case a minimum bound on the number of interactions should be found. Moreover, once such a bound defined, the scalability of reachability analysis becomes an issue (mainly due to the magnitude of such a numeric bound). In the context of the LGS, the CaV approach, along with the PastFree[ze] algorithm, offers the tools to address these challenges by focusing on the environment model. To find a minimum bound on the number of pilot interactions, we have arbitrarily fixed it to 1000 and while running the reachability analysis in the nominal mode we have observed that after 7 Handle commands the size of the clusters induced by the state of environment starts repeating. Table III shows the cardinality of the state-space at each environment step, it should be noted that a pattern emerges ($H_{2n} = H_{2n+2}$ and that $H_{2n+1} = H_{2n+3}$, where $n \in [3..498]$), which holds for the next 994 environment/SUS interactions. This pattern provides a strong indication of the cyclicality of the system modulo the environment model, which can be

Prepared using stvrauth.cls
DOI: 10.1002/stvr

Table III. Cardinality of the clusters at each context step (progress of the environment)

Context step	Before init	After init	H1	H2	H3	H4	H5	H6	H7	H8	H9
Cluster size	1 326	1	448	12 904	44 331	72 650	85 042	87 780	87 746	87 780	87 746

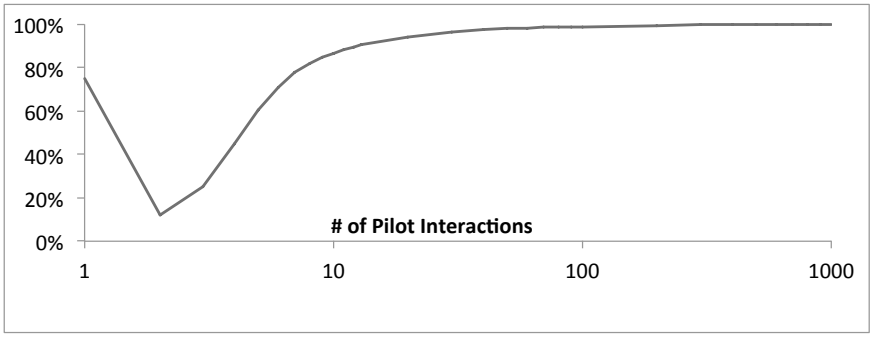


Figure 15. Percentage of the state-space freed due to PastFree[ze] - 1 to 1000 handle occurrences

proved by bisimulation on the two state spaces induced by the environment. For the LGS composed with a cyclic Pilot scenario the reachability diameter r_d is 305 (the value was computed on the state-space of the composition). By composing the LGS with a Pilot scenario unrolled to a depth $b_{is} = 8$ the same reachability diameter is achieved that proves the completeness of the analysis. Based on these encouraging results, in the future we plan to integrate the Past-Free[ze] algorithm into an automated algorithm for proving the completeness of the analysis.

To show the scalability of our approach, Fig 15 shows the percentage of the state-space that can be freed from memory for a varying number of pilot interactions (1 Handle, up to 1000 Handles) when using the PastFree[ze] context-driven reachability algorithm. For a low number handles (less than 7) the large number of states in the last context steps (see Table III) lowers the ratio of freed states over total states. However, starting from 8 handles the ratio exceeds 80% of the state-space during each run, increasing up to 99.9% for 1000 handles. In the case of an arbitrary finite number of handles, at any given time during the analysis, the PastFree[ze] algorithm keeps in memory only the state-space cluster induced by the current environment state and the one corresponding to the next one. Hence the scalability is only bounded by the size of the two clusters and an eventual time-limit (the larger the number of Handles commands send the longer it takes to explore all the state-space), and not by the size of the complete state-space (as is the case with other state-of-the-art reachability algorithms). For instance, using BFS analysis all cases with more than 400 handles fail on L64 due to the lack of memory. Figure 16 presents the size of the state-space for 1 to 1000 handle interactions and the L64 explosion limit. For 1000 handles the size of the state space approaches 80GB (87 540 904 states). Using our context-driven reachability algorithm the analysis of all 1000 cases was successful using less than 10GB of RAM, which represents only 15% of the total amount of memory available on $L64$.

6. RELATED WORK

Since the introduction of model-checking in the early 1980s [36], several model-checker tools have been developed to help the verification of concurrent systems [4, 29, 44].

To enable the verification of ever larger systems, numerous research efforts are focused on reducing the impact of the state-space explosion problem.

Some of these approaches prune the state-space using techniques such as partial-order reduction [27, 35, 41] and symmetry reduction [14] that exploit fine-grain transition interleaving symmetries and global system symmetries respectively. Our approach is complementary to such techniques, focusing on the topological relations between the system-states instead of their symmetries.

DOI: 10.1002/stvr

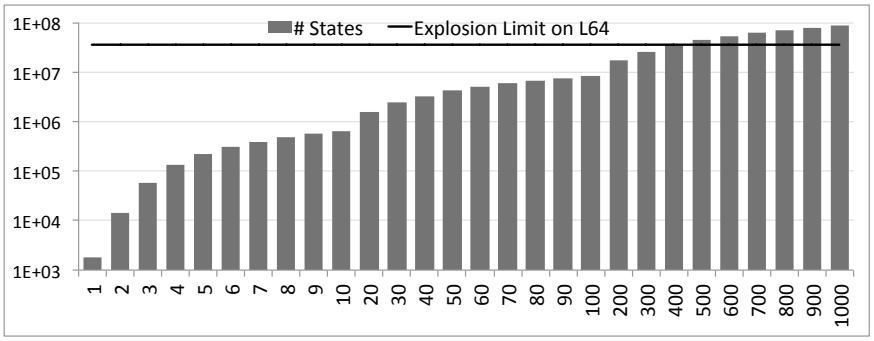


Figure 16. Reachability results for 1 to 1000 pilot interactions in the nominal mode.

Yet other approaches, such as TLC [44] or semi-external LTL model-checking [2, 3, 24], focus on algorithmic advancements and the maximal use of the available resources such as external memories(disk). The Past-Free[ze] algorithm, presented in this study, is also a semi-external algorithm (the data-structure is distributed in memory and on disk). However, as opposed to these approaches, our algorithm uses the disk only for enabling the construction of counter-examples. As such, during the reachability, the configurations of past-clusters are simply written to disk without the need to read them. At the end of the analysis, if a property is violated, the disk-stored configuration are read to compute the counter-example, which will be presented to the user. At the moment, our technique focuses only on the reachability analysis and not on the whole model-checking problem as in [24]. Nevertheless, besides the need for defining an I/O efficient state-matching strategy, there is no practical limitation preventing the use of techniques such as those presented in [24] to enable the offline LTL model checking in conjunction with the Past-Free[ze] algorithm.

Techniques such as bounded model-checking [11] (BMC) exploit the observation that in many practical settings the property verification can be done with only a partial (bounded) reachability analysis. Hence, in the absence of a full-coverage proof, these approaches cannot guarantee the absence of errors, but only their presence. The usage of explicit acyclic behaviors, and the CaV approach can be considered as the explicit-state equivalent of symbolic BMC. Moreover, as opposed to BMC, the usage of acyclic behaviors offers more flexibility for specifying the ”bounds” of the analysis, and the context can be seen as a high-level skeleton which drives the analysis through a complex state-space partition.

In the context of symbolic model-checking, the verification problem is encoded as a boolean equation and a satisfiability procedure is used for model-checking. In this context, numerous research efforts [22, 28, 30] focus on the problem of partitioning one SAT instance into multiple smaller instances that are easier to solve potentially in parallel. The recursive decomposition strategy used in CaV (Section 3.2) can be seen as an instance of such a partitioning strategy. However, instead of relying on the potentially complex relations between the system variables, the context split exploits the acyclicity of the interaction scenario to recursively partition it up to linear paths.

While the previous techniques address the property verification problem monolithically, compositional verification [26] focuses on the analysis of individual components of the system using assume/guarantee reasoning to extract (sometimes automatically) the interactions that a component has with its environment and to reduce the model-checking problem to these interactions. Once each individual component is proved correct the composition is performed using operators that preserve the correctness. The CaV approach can be seen as a coarse-grain compositional verification, where the focus is steered towards the interactions of the whole system with its surrounding environment (context).

Conversely to traditional techniques in which the surrounding environment is often implicitly modeled in the system (to obtain a closed system), in CaV the environment is explicitly specified isolated from the model. In this context the technique described in this paper, offers one more tool

Prepared using stvrauth.cls
DOI: 10.1002/stvr

besides the automatic context splitting [17] for pushing the state-space explosion limits and enabling full-system verification on real-world industrial systems.

7. CONCLUSIONS

In this paper, we presented a new exhaustive reachability analysis algorithm that directly addresses the state-space explosion problem. This algorithm, named Past-Free[ze], relies on the isolation of the acyclic behaviors of the system and the use of their graph-theoretic properties to reduce the memory requirements during reachability analysis. For modeling the acyclic behaviors we adopted the CaV technique which offers the tools for their identification, isolation and formal specification. Our approach was implemented in the OBP *Observation Engine* and was evaluated both on a synthetic benchmark of 50 test cases, and on two industrial-size verification problems. The results show that much of the state-space can be freed from memory during analysis, enabling the exploration of much larger systems compared to other approaches. The run-time performances of the reachability algorithms are improved by the use of the configuration clusters induced by the acyclic LTS. Moreover, in one of the realistic case studies Past-Free[ze] enabled the analysis of arbitrarily large state-spaces by relying on the topological properties of the environment model captured using the CDL language.

While the Past-Free[ze] technique effectively enabled the exploration of larger systems, the need to store the state-space (on disk) for offline processing (counter-example construction, offline LTL verification, etc) adds a time overhead due to the I/O operations. Moreover, the evaluation of the timing characteristics of our approach was hindered by the interferences with the Java object-memory management routines. We are currently investigating different techniques that can reduce the I/O overhead, and we are focusing on more sophisticated object-memory schemes that will reduce the interferences with the platform and will render the run-time analysis more accurate.

The reachability algorithm, presented in this paper, exploits the structural properties of the systems subject to model-checking with promising results. Some future research directions are:

- The core of the Past-Free[ze] algorithm is based on the topological order of a DAG, and since this order is not unique and rather arbitrary we plan to investigate the impact different linear orders can have on the performances of our approach.

- The integration with semi-external model-checking algorithms, such as [2, 3, 24], which would extends the applicability of Past-Free[ze] for problems with large acyclic induced clusters and/or with a large number of future clusters (a high ReachedFuture value).

- The study of the complementarity of our Past-Free[ze] algorithm with other state-space reduction techniques, such as partial-order reduction, for improving the scalability of model-checking on large industrial case studies.

REFERENCES

1. Yamine Aït Ameur and Klaus-Dieter Schewe, editors. *Abstract State Machines, Alloy, B, TLA, VDM, and Z - 4th International Conference, ABZ 2014, Toulouse, France, June 2-6, 2014. Proceedings*, volume 8477 of *Lecture Notes in Computer Science*. Springer, 2014.
2. J. Barnat, L. Brim, P. Šimeček, and M. Weber. *Tools and Algorithms for the Construction and Analysis of Systems: 14th International Conference, TACAS 2008, Held as Part of the Joint European Conferences on Theory and Practice of Software, ETAPS 2008, Budapest, Hungary, March 29-April 6, 2008. Proceedings*, chapter Revisiting Resistance Speeds Up I/O-Efficient LTL Model Checking, pages 48–62. Springer Berlin Heidelberg, Berlin, Heidelberg, 2008.
3. Jiri Barnat, Lubos Brim, and Pavel Simecek. Cluster-based i/o-efficient ltl model checking. In *Proceedings of the 2009 IEEE/ACM International Conference on Automated Software Engineering*, ASE '09, pages 635–639, Washington, DC, USA, 2009. IEEE Computer Society.
4. Johan Bengtsson, Kim G. Larsen, Fredrik Larsson, Paul Pettersson, and Wang Yi. UPPAAL — a Tool Suite for Automatic Verification of Real–Time Systems. In *Proc. of Workshop on Verification and Control of Hybrid Systems III*, number 1066 in Lecture Notes in Computer Science, pages 232–243. Springer–Verlag, October 1995.

DOI: 10.1002/stvr

5. Johan Bengtsson and Wang Yi. *Lectures on Concurrency and Petri Nets: Advances in Petri Nets*, chapter Timed Automata: Semantics, Algorithms and Tools, pages 87–124. Springer Berlin Heidelberg, Berlin, Heidelberg, 2004.
6. B. Berthomieu, P.-O. Ribet, and F. Verdanat. The tool TINA - Construction of Abstract State Spaces for Petri Nets and Time Petri Nets. *International Journal of Production Research*, 42:2741–2756, July 2004.
7. Bernard Berthomieu, Silvano Dal Zilio, and Lukasz Fronc. Model-Checking Real-Time Properties of an Aircraft Landing Gear System Using Fiacre. In *4th International ABZ Conference*, volume 433, pages 110–125, France, June 2014. Springer.
8. Frédéric Boniol, Philippe Dhaussy, Luka Le Roux, and Jean-Charles Roger. *Model-Based Analysis*, pages 157–184. Wiley, May 2013.
9. Frédéric Boniol, Virginie Wiels, Yamine Ait Ameur, and Klaus-Dieter Schewe, editors. *Context-Aware Verification of a Landing Gear System*, volume 433 of *Communications in Computer and Information Science*. Springer International Publishing, 2014.
10. J. R. Burch, E. M. Clarke, K. L. McMillan, D. L. Dill, and L. J. Hwang. Symbolic model checking: 10^{20} states and beyond. In *5th IEEE Symposium on Logic in Computer Science*, pages 428–439, 1990.
11. Edmund Clarke, Armin Biere, Richard Raimi, and Yunshan Zhu. Bounded model checking using satisfiability solving. *Formal Methods in System Design*, 19(1):7–34, 2001.
12. EdmundM. Clarke and E.Allen Emerson. Design and synthesis of synchronization skeletons using branching time temporal logic. In Dexter Kozen, editor, *Logics of Programs*, volume 131 of *Lecture Notes in Computer Science*, pages 52–71. Springer Berlin Heidelberg, 1982.
13. E.M. Clarke, E.A. Emerson, and A.P. Sistla. Automatic verification of finite-state concurrent systems using temporal logic specifications. *ACM Trans. Program. Lang. Syst.*, 8(2):244–263, 1986.
14. E.M. Clarke, R. Enders, T. Filkorn, and S. Jha. Exploiting symmetry in temporal logic model checking. *Formal Methods in System Design*, 9(1-2):77–104, 1996.
15. Thomas H. Cormen, Charles E. Leiserson, Ronald L. Rivest, and Clifford Stein. *Introduction to Algorithms, Third Edition*. The MIT Press, 3rd edition, 2009.
16. P. Dhaussy, Frédéric Boniol, and Jean-Charles Roger. Context aware model-checking for embedded software. In *Embedded System / Book 1*. Intech publisher, 2012.
17. Philippe Dhaussy, Frédéric Boniol, and Jean-Charles Roger. Reducing state explosion with context modeling for model-checking. In *13th IEEE International High Assurance Systems Engineering Symposium (Hase'11)*, Boca Raton, USA, 2011.
18. Philippe Dhaussy, Frédéric Boniol, Jean-Charles Roger, and Luka Leroux. Improving model checking with context modelling. *Advances in Software Engineering*, ID 547157:13 pages, 2012.
19. Philippe Dhaussy, Frédéric Boniol, Jean-Charles Roger, and Luka Leroux. Improving model checking with context modelling. *Adv. Soft. Eng.*, 2012:9:9–9:9, January 2012.
20. Philippe Dhaussy, Pierre-Yves Pillain, Stephen Creff, Amine Raji, Yves Le Traon, and Benoit Baudry. Evaluating context descriptions and property definition patterns for software formal validation. In Bran Selic Andy Schuerr, editor, *12th IEEE/ACM conf. Model Driven Engineering Languages and Systems (Models'09)*, volume 5795, pages 438–452. Springer-Verlag, LNCS, 2009.
21. Philippe Dhaussy, Jean-Charles Roger, Luka Leroux, LabSticc ENSTA Bretagne, Brest France, and Frédéric Boniol. Context aware model exploration with obp tool to improve model-checking. *Embedded Real-Time Software and Systems (ERTS'12)*, 2012.
22. V. Durairaj and P. Kalla. Guiding cnf-sat search via efficient constraint partitioning. In *Computer Aided Design, 2004. ICCAD-2004. IEEE/ACM International Conference on*, pages 498–501, Nov 2004.
23. Matthew B. Dwyer, George S. Avrunin, and James C. Corbett. Property specification patterns for finite-state verification. In *FMSP*, pages 7–15, 1998.
24. Stefan Edelkamp, Peter Sanders, and Pavel Šimeček. Semi-external ltl model checking. In Aarti Gupta and Sharad Malik, editors, *Computer Aided Verification*, volume 5123 of *Lecture Notes in Computer Science*, pages 530–542. Springer Berlin Heidelberg, 2008.
25. Patrick Farail, Pierre Gaufillet, Florent Peres, Jean-Paul Bodeveix, Mamoun Filali, Bernard Berthomieu, Saad Rodrigo, Francois Vernadat, Hubert Garavel, and Frédéric Lang. FIACRE: an intermediate language for model verification in the TOPCASED environment. In *European Congress on Embedded Real-Time Software (ERTS)*, Toulouse, january 2008. SEE.
26. Cormac Flanagan and Shaz Qadeer. Thread-modular model checking. In *SPIN'03*, 2003.
27. P. Godefroid. The Ulg partial-order package for SPIN. *SPIN Workshop*, 1995.
28. A. Gupta, Zijiang Yang, P. Ashar, Lintao Zhang, and S. Malik. Partition-based decision heuristics for image computation using sat and bdds. In *Computer Aided Design, 2001. ICCAD 2001. IEEE/ACM International Conference on*, pages 286–292, Nov 2001.
29. G.J. Holzmann. The model checker SPIN. *Software Engineering*, 23(5):279–295, 1997.
30. Antti E. J. Hyvärinen, Tommi Junttila, and Ilkka Niemelä. *Logic for Programming, Artificial Intelligence, and Reasoning: 17th International Conference, LPAR-17, Yogyakarta, Indonesia, October 10-15, 2010. Proceedings*, chapter Partitioning SAT Instances for Distributed Solving, pages 372–386. Springer Berlin Heidelberg, Berlin, Heidelberg, 2010.
31. Daniel Kroening and Ofer Strichman. *Verification, Model Checking, and Abstract Interpretation: 4th International Conference, VMCAI 2003 NewYork, NY, USA, January 9–11, 2003 Proceedings*, chapter Efficient Computation of Recurrence Diameters, pages 298–309. Springer Berlin Heidelberg, Berlin, Heidelberg, 2003.
32. Pavel Parizek and Frantisek Plasil. Specification and generation of environment for model checking of software components. *Electr. Notes Theor. Comput. Sci.*, 176(2):143–154, 2007.
33. Sachoun Park and Gihwon Kwon. Avoidance of state explosion using dependency analysis in model checking control flow model. In *Proceedings of the 5th International Conference on Computational Science and Its Applications (ICCSA '06)*, volume 3984, pages 905–911. Springer-Verlag, LNCS, 2006.

DOI: 10.1002/stvr

34. Radek Pelánek. Beem: Benchmarks for explicit model checkers. In *Proceedings of the 14th International SPIN Conference on Model Checking Software*, pages 263–267, Berlin, Heidelberg, 2007. Springer-Verlag.
35. D. Peled. Combining Partial-Order Reductions with On-the-fly Model-Checking. In *CAV '94: Proceedings of the 6th International Conference on Computer Aided Verification*, pages 377–390, London, UK, 1994. Springer-Verlag.
36. Jean-Pierre Queille and Joseph Sifakis. Specification and verification of concurrent systems in cesar. In *Proceedings of the 5th Colloquium on International Symposium on Programming*, pages 337–351, London, UK, 1982. Springer-Verlag.
37. MichelA. Reniers and TimA.C. Willemse. Folk theorems on the correspondence between state-based and event-based systems. In Ivana Černá, Tibor Gyimóthy, Juraj Hromkovič, Keith Jefferey, Rastislav Královič, Marko Vukolić, and Stefan Wolf, editors, *SOFSEM 2011: Theory and Practice of Computer Science*, volume 6543 of *Lecture Notes in Computer Science*, pages 494–505. Springer Berlin Heidelberg, 2011.
38. Ciprian Teodorov, Philippe Dhaussy, and Luka Roux. Environment-driven reachability for timed systems. *International Journal on Software Tools for Technology Transfer*, pages 1–17, 2015.
39. Ciprian Teodorov, Luka Leroux, and Philippe Dhaussy. Context-aware verification of a cruise-control system. *to appear in 4th International Conference on Model & Data Engineering (MEDI'2014)*, 2014.
40. Oksana Tkachuk and Matthew B. Dwyer. Environment generation for validating event-driven software using model checking. *IET Software*, 4(3):194–209, 2010.
41. Antti Valmari. Stubborn sets for reduced state space generation. In *Proceedings of the 10th International Conference on Applications and Theory of Petri Nets*, pages 491–515, London, UK, 1991. Springer-Verlag.
42. Jon Whittle. Specifying precise use cases with use case charts. In *9th IEEE/ACM conf. Model Driven Engineering Languages and Systems (MoDELS'06), Satellite Events*, pages 290–301, Genova, Italy, October 1-6 2006.
43. Kenro Yatake and Toshiaki Aoki. Automatic generation of model checking scripts based on environment modeling. In *Model Checking Software - 17th International SPIN Workshop, Enschede, The Netherlands, September 27-29, 2010. Proceedings*, pages 58–75, 2010.
44. Yuan Yu, Panagiotis Manolios, and Leslie Lamport. Model checking tla+ specifications. In *Proceedings of the 10th IFIP WG 10.5 Advanced Research Working Conference on Correct Hardware Design and Verification Methods*, CHARME '99, pages 54–66, London, UK, UK, 1999. Springer-Verlag.

DOI: 10.1002/stvr