Int J Softw Tools Technol Transfer
DOI 10.1007/s10009-015-0401-2

ABZ 2014

Environment-driven reachability for timed systems

Safety verification of an aircraft landing gear system

Ciprian Teodorov[1] · Philippe Dhaussy[1] · Luka Le Roux[1]

Abstract With an ever increasing complexity, the verification of critical embedded systems is a challenging and expensive task. Among the available formal methods, model checking offers a high level of automation and would thus lower the cost of this process. But, the scalability of this technique is hindered by the state-space explosion problem, which fuelled the research community since its inception. To address this challenge, in the case of real size systems, the theoretical, the methodological and the algorithmic axes have to be integrated. The context-aware verification (CaV) strives to do this by focusing on the identification, the isolation and the reification of the environment surrounding the studied system. It enables the use of specific algorithms with a major, positive, impact on the scalability of model checking. In this paper, we apply this technique to study a Landing Gear System (LGS) in the presence of failures. The problem has been decomposed in 885 independent verification units (called contexts). The analysis of 163 of these contexts on a 64 GB computer unraveled a 20 TB state space with more than 2.2 billion states. Moreover, using this approach arbitrarily long scenarios have been analysed using less than 10 GB of memory.

Keywords Formal verification · Context-aware verification · Reachability analysis

✉ Ciprian Teodorov
ciprian.teodorov@ensta-bretagne.fr

1 UEB, Lab-STICC Laboratory UMR CNRS 6285, ENSTA Bretagne, 2 rue François Verny, 29806 Brest, France

1 Introduction

Nowadays, the verification phase of aerospace control systems is one of the most expensive phases of the development process, due to the complexity and the safety requirements. Manufacturers of industrial systems make significant efforts in testing and simulation to successfully pass the certification processes. The industry moves towards the integration of more and more autonomous components, such as automated mission planning, multivehicle cooperative control, automatic collision avoidance. These tendencies will further increase the cost of the verification phase exponentially.

Formal methods provide an alternative to traditional test-driven verification [27]; however, their scalability to an industrial process poses numerous challenges. For instance, model checking is an exhaustive-search technique that enables to automatically prove if a system satisfies the requirements. Moreover, for each requirement that is not satisfied a counter-example is produced. But, for large systems, due to the internal complexity and the high-degree of concurrency, this technique can lead to an unmanageable large number of possible behaviors. This problem, known as the state-space explosion problem [11,31,39], has fuelled numerous research efforts. The invention of numerous techniques, ranging from algorithm optimizations [9,25,34], to model simplifications [12,24,32,38] and decomposition methods [23], has pushed the limits of model checking ever further. Nevertheless, for successful verification, in the case of large and complex systems, the designers still have to manually tune the "verification model" to restrict its behaviors. This process is tedious, error prone and poses a number of methodological challenges since different versions of the model should be kept consistent, in sync and maintained. The Context-aware Verification (CaV) [17], overviewed in Sect. 2, provides a structured approach for addressing some

Published online: 23 September 2015

of these problems. With CaV, a number of independent *verification contexts* are used to represent the restricted model behaviors along with the associated requirements. These contexts, specified using the CDL language (introduced in Sect. 2.1), are exploited by the verification tools giving rise to new algorithms for fighting the state-space explosion problem. Two such algorithms, a context-driven reachability algorithm (named PastFree[ze]), and an automated context partitioning method, are presented in Sect. 2.2.

Fully supported by the OBP *Observation Engine*[1], the CaV approach is complementary to the state-of-art research in the model-checking field (discussed in Sect. 5) and enables at least three different state-space reduction axes:

– the environment can be decomposed in contexts, thus isolating different operating modes of the system;
– each context can be exploited by the analysis tools, for instance by automated partitioning in independent verification problems;
– the requirements, which are associated with each context, are focused on specific environmental conditions.

In this study, we present a detailed account of applying the CaV method to an aircraft landing gear system (LGS). Initially studied by Boniol et al. [8], this case study was proposed by Boniol and Wiels as a challenge for a dedicated track at the 4th International ABZ Conference (ABZ'14) [7].

The LGS is modeled using the Fiacre language [22] (Sect. 3.1). The environment is composed of pilot interactions (requesting the extension or retraction of the landing gear) interleaved with failure injections. One top-level interaction scenario captures the full environment, but bounds the number of pilot interactions to an arbitrary value. Section 3.2 overviews the CDL specification of the environment. Since this study is focused more on the reachability analysis of the LGS using the CaV approach, only two safety properties are considered (Sect. 3.3). Both these properties are associated with the top-level environment to create the global verification context.

The global verification context is decomposed into 885 smaller contexts, each one representing a different failure sequence setup. The verification iteratively performs reachability analysis on each of them. The results presented in Sect. 4 overview the reachability of 163 of the 885 contexts considering only the full LGS configuration with three gears and doors. The results previously presented by the authors at ABZ'14 [19] are updated and the differences are briefly discussed in Sect. 4.2.

The 163 contexts correspond to the 18 one failure contexts, 64 of the two failure contexts, and 81 of the three failure contexts. The successful exploration of all the one failure contexts is presented in Sect. 4.3. The results for the two and three failure contexts are overviewed in Sect. 4.4 and emphasize the importance of the context reification[2] during the reachability analysis. The impact of using infinitely many or a bounded number of pilot interactions is discussed in Sect. 4.5 mainly showing the importance of having a reified environment model. One of the most important result presented is the possibility to analyse arbitrarily many pilot interactions with constant memory requirements.

Overall, 2.2 billion states of the LGS were analysed in around 18 % of the total number of contexts. By extrapolating the results obtained for the 163 context to the total number scenarios, we could say that, for the LGS case study, the CaV context-aware partitioning enables the verification of a state space of around 20TB on a 64 GB computer (a 300× gain).

The first experiences on environment reification [17], the formalisation of the CDL language [15] and the introduction of the context-aware partitioning technique [16,18] led to realistic case studies from different application domains, such as healthcare (pacemaker [6]), aeronautics (LGS [19]), and automotive (cruise-control [36]). Building on these developments, the main contributions of this study are:

– A new reachability algorithm, named PastFree[ze], which enables the analysis of arbitrarily large interaction scenarios with low-memory requirements.
– A synthetical presentation of the context-aware verification approach, which emphasizes the importance of independent verification units.
– The reachability analysis of a realistic case study from the aeronautic industry, which shows good results regarding the scalability of our approach.

These results are very promising, and trading off memory usage for an increased number of contexts has two advantages. Firstly, large verifications succeed where previously they could not. Secondly, the tremendous advancements of the cloud-computing industry offer the opportunity to exploit large numbers of computational resources having relatively low amounts of available memory [3].

2 Context-aware verification

Model checking is a technique that relies on building a finite model of a system of interest, and checking that a desired property, typically specified as a temporal logic formula, holds for that model. Since the introduction of this technology in the early 1980s [33], several model-checker tools have

[1] OBP Observation Engine website: http://www.obpcdl.org.

[2] By "reification", we mean the process of explicitly identifying something, which then is formulated and rendered accessible to conceptual/computational manipulation.

been developed to help the verification of concurrent systems [4,26].

However, while model checking provides a rigorous and automated framework for formal system verification and has successfully been applied on industrial systems, it suffers from the state-space explosion problem [39]. This is due to the exponential growth of the number of states the system can reach with respect to the number of interacting components. Since its introduction, model checking has progressed significantly, with numerous research efforts focused on reducing the impact of this problem [9,12,24,38], thus enabling the verification of ever larger systems. Besides these techniques, in the case of large and complex system, the system designers manually tune the "verification model" to restrict its behaviors to the ones pertinent relative to the specified requirements. This process is tedious, error prone and poses a number of methodological challenges since different versions of the model should be kept sound, synchronized 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 (Sect. 2.1). The verification contexts are created by associating with 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.

This user-defined environment decomposition is mainly intended to capture the operating modes of the SUS, and the different verification objectives for each operating mode. Moreover, such a user-guided environment decomposition is very important in the presence of properties requiring the storage of the execution history besides the configuration of the system (ex. observer automata). In such cases, a large number of properties can significantly increase the size of the state space leading to the state-space explosion. The verification engineer can in such cases create multiple verification units with identical interaction scenarios associated with different verification targets.

The CaV approach imposes a formal, methodical decomposition and classification of large requirements sets, a first step in overcoming the state-space explosion problem. The uniqueness of this method lies in the way it exploits the contexts during the analysis to push the limits of the verifications even further. In the following, we briefly overview a context-directed reachability algorithm (Sect. 2.2.1), and a partitioning algorithm that enables the automated decomposition of contexts (Sect. 2.2.2). The effectiveness of these techniques, for the LGS case study, is discussed in the results section (Sect. 4). From a system-engineering point of view, this approach solves the methodological issues by decoupling the SUS from its environment, thus allowing their refinement in isolation. Furthermore, the possibility to analyse the SUS with a partial environment model gives valuable insights on particular context-dependent behaviors, enabling the designers to better focus their verification efforts. The reader should note the difference between the used-defined verification units (primarily intended to capture the verification of groups of system-level behaviors, eg. operating modes) and the creation of sub-contexts through the automated context decomposition (used to improve the scalability of model checking in the case of state-space explosion). While a full CaV methodology is very important for the effective industrialisation of the approach, an in-depth discussion on this subject is beyond the scope of this study, which is focused on the impact of explicit context isolation on the reachability problem.

The context-aware verification approach is implemented in the OBP *Observation Engine*[16] (overviewed in Sect. 2.3), which is publicly available.[3]

2.1 Verification contexts with CDL formalism

In the context of reactive embedded systems, the environment of each component of the system is often well known. It is, therefore, better and more effective to identify and express this environment than trying to reduce the state space of the SUS (by ad-hoc modifications). However, it should be noted that the formal relevance of this approach is based on the following hypothesis: It is possible to specify the sets of bounded behaviors in a complete way. Even though this can be seen as a limiting hypothesis, it expresses no more than the following well accepted idea: A software system can be correctly developed only if we know the constraints of its use. Thus, we take for granted that the designer is able to identify the perimeter (constraints, conditions) of the SUS and all possible interactions between it and its environment. Another important observation is that the properties are often related to specific use cases (such as initialization, reconfiguration, degraded modes). Therefore, it is not necessary for a given property to take into account all possible behaviors of the environment, but only the ones concerned by its verification.

To formalize the context specification in [15], we introduced the CDL formal language to capture the interactions with the environment. The following sections overview the CDL specification of these interaction-based scenarios (Sect. 2.1.1), the property language for capturing the

[3] OBP Observation Engine website: http://www.obpcdl.org.

requirements (Sect. 2.1.2), and the top-level structure of a verification context (2.1.3). The reader should refer to [15] for an in-depth presentation of the CDL language semantics.

2.1.1 Environment modelling through interaction scenarios

For expressing the environment interactions with the SUS, the CDL language provides a textual syntax based on the Use-Case Charts [40] using activity and sequence diagrams, which have been extended to permit the description of several entities (actors). The result is a finite generalized Message Sequence Chart (MSC) C expressed using the following formal grammar:

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

In other words, the environment interactions with the SUS are captured through either: (1) a sequence of emissions $a!$ and receptions $a?$, or (2) a sequential composition (seq denoted ;) of two MSCs ($C_1; C_2$), or (3) a non-deterministic alternative (alt denoted $[]$[4]) between two MSCs ($C_1[]C_2$), or (4) a parallel composition (par denoted $\|$) between two MSCs ($C_1\|C_2$).

An emission is an asynchronous communication from the environment to the SUS. Similarly, a reception is an asynchronous communication from the SUS to the environment. These interactions are expressed using the following grammar:

$$\begin{aligned} a! &::= send\ Value\ to\ \{p\}n \\ a? &::= receive\ (Value|any)\ from\ \{p\}n \end{aligned}$$

The Value non-terminal represents a value expressed in the SUS modelling language (Fiacre in this study). The special value "any" defines a reception that ignores the actual value sent from the SUS. $\{p\}n$ refers to the nth instance of the process named p.

For example, in the case of the LGS, we define a manipulation of the handle by the pilot as an emission:
event Handle **is** { **send** HANDLE **to** {Dispatcher}1}.

The hierarchical composition of M is introduced by the $activity$ keyword. For instance, the interleaving of 2 Handle occurrences with one failure can be expressed through the following activities:
activity 2handles **is** { loop 2 Handle }
activity handleAndFailure **is** { 2handles || injectFailure },
where injectFailure is an emission interaction. Note that $loop\ n$ construct is only syntactic sugar for expressing chains of sequential interactions using iteration (bounded loops).

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

2.1.2 Property specification

The CDL formalism provides 3 distinct 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 nothing more than propositional logic formulas, which could be either asserted in a context, or be used in the definition of observers or property patterns. The predicates are expressed using the following formal grammar:

$$P ::= Atom \mid not\ P \mid P\ and\ P$$

For example, **predicate** pRed **is** { {proc}1:red_light=true } expresses that the variable red_light of $\{proc\}1$ is true. In this case, the expression {proc}1:red_light=true is an atomic side-effect free expression (Atom non-terminal) expressed in terms of the structure of the configuration of the system. The other propositional logic operators can be trivially obtained using the DeMorgan's laws.

The observers are timed automata-based constructs 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.

In the CDL formalism, the observers are expressed using the following formal syntax:

$$O ::= (clk+)?(id_s\ C_P?\ E?\ P?\ (R_{clk}+)?\ (id_t|reject))+$$

where the $clk+$ part introduces a number of clock variables. A transition between the states id_s and id_t is fireable if the clock invariants C_P are true, the events E are present and the predicates P are true, in which case the transition is executed potentially resetting the clocks R_{clk}. The notion of $clock$ present in the observers implements standard timed automata semantics [1]. The notion of "event" (E) is used to capture the changes in predicate valuation (rising/falling edges). Their syntax is

$$E ::= (P\ becomes\ (true\,|\,false)) \mid (P\ changes)$$

A property encoded by an observer is violated if, during the execution of the system, the observer reaches a predefined rejection state ($reject$). The syntactic ordering of the transition ensures the determinism of the observers. Their completeness is mechanically guaranteed by the exploration engine by implicit transitions looping in each state (complementing the explicitly defined transitions). In the concrete syntax, the clock invariants, the predicates and the clock reset blocks are syntactically expressed between "–" and "->" transition markers, and are separated amongst themselves by the "/" character.

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** 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 [20]. The interested reader should refer to [18] for more details on this aspect, which is beyond the scope of this study.

2.1.3 The anatomy of a verification context

The interaction scenarios, the properties and their associations are explicitly defined by the system designer, which is responsible for their pertinence and their completeness with respect to the SUS requirements. In the CDL language, the association of interaction-based scenarios with the relevant properties form well-defined verification units, named CDL contexts (or simply contexts). The formal structure of a context conforms with the following syntax:

$$CDL ::= P* \ O* \ C_{init}? \, C_{main}?$$

A context, introduced through the *cdl* keyword, is structured into two distinct sections: the *property assertion* part, and the *scenario specification* part. The *property assertion* part contains the (potentially empty) list of predicates ($P*$) that should be globally satisfied (should be true in all states of the state space), and the list of observers ($O*$) 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. It should be noted that both the $init$ and the $main$ blocks can be empty. Leaving both the initialisation and core scenario empty, and specifying the environment within the same formalism as the SUS, leads to the traditional model-checking setup.

2.2 Exploiting the verification contexts

As already mentioned, the uniqueness of the CaV approach lies in the way the contexts are used during the verification run. In this section, we overview a reachability algorithm (Sect. 2.2.1), and a context partitioning algorithm (Sect. 2.2.2), which exploit the acyclicity of the context to enable the successful analysis of large verification problems.

2.2.1 PastFree[ze]: context-directed reachability

One of the most important properties enforced by the CDL context is that the interaction scenario is acyclic. The reachability algorithms can then use this particularity to drive the analysis. The PastFree[ze] algorithm builds on this and 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 [13], and can be formally defined as a linear order between the vertices of a DAG (the states of the context scenario in our case) such that if there exists a transition $u \rightarrow v$ between two states u, v then u is present before v in the ordering, expressed as $u < v$ (u precedes v, or u is an ancestor of v). Figure 1b shows one topological order of the DAG in Fig. 1a. Such an ordering can be constructed in linear time with respect to the size of the DAG.

Relying on the *topological ordering*, the reachability algorithm can "*forget the past to focus on the future*". Practically, this means that if the LTS states are indexed according to the context 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.

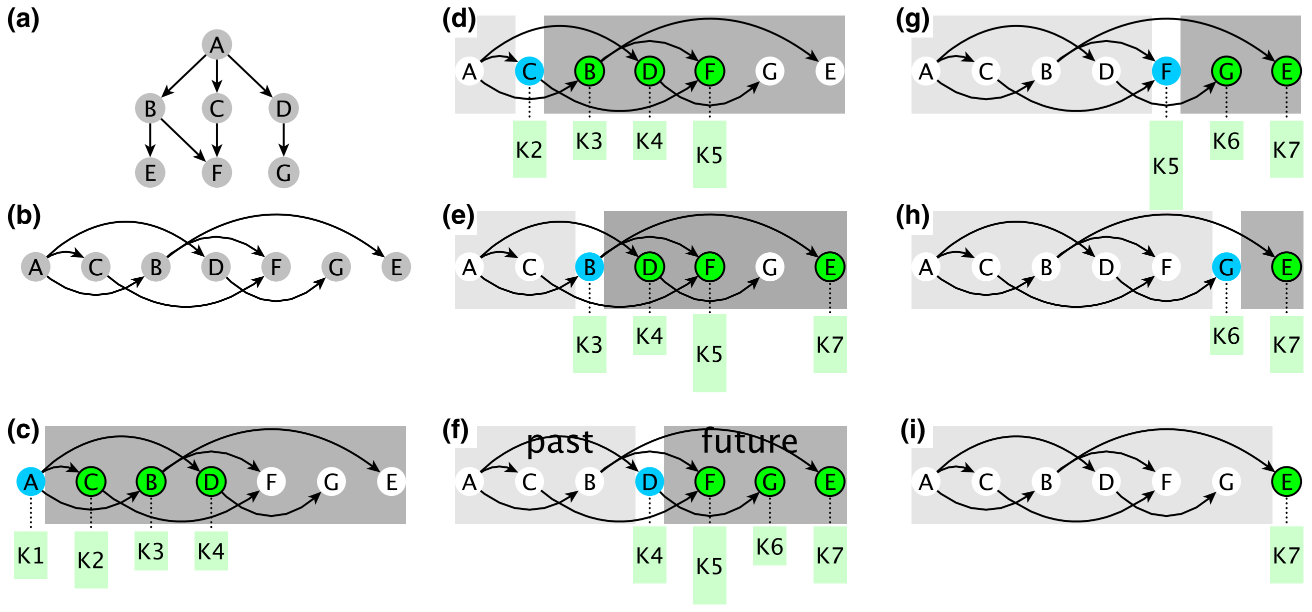

Fig. 1 Illustration of the PastFree[ze] reachability algorithm. **a** Acyclic interaction scenario T_a; **b** topological ordering of T_a; **c**–**i** reachability analysis of $T_c||T_a$. The K1–K7 rectangles represent clusters of configurations (K_i), their height being proportional to the number of configuration stored. The *white nodes* represent either past configuration clusters (freed from memory) or future configuration clusters not reached yet (not present in memory). *Blue nodes* between the shaded areas represent the cluster being analysed, while the *green nodes* with black circles around are live clusters present in memory (either reached in the past, or reachable from the current cluster)

The context induces a clustering of the space of configurations. Each cluster i is identified by the state of the context and contains the set of configurations $\langle -, i\rangle$, where $-$ denotes any state of the SUS. From this perspective, the PastFree[ze] algorithm 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, lets take for example the context-induced DAG (T_a) in Fig. 1a composed with an arbitrary SUS (T_c)—T_c can be cyclic. One topological ordering of T_a is shown in Fig. 1b. Using this ordering, the reachability analysis (Fig. 1c) starts by processing all configurations reachable from the initial state $\langle s_0, A\rangle$. When a transition from T_c is fired, the resulting configurations will be elements of the cluster $\langle -, A\rangle$. 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 freed from memory (see Fig. 1d). This process repeats until the last configuration from cluster $\langle -, E\rangle$ is analysed, at which point the analysis ends (see Fig. 1i).

Freeing the past-state clusters from memory, effectively decreases the memory pressure during reachability. However, if the verification run fails, it is impossible to build a counter-example exposing a path from the initial configuration to the failure point. In these cases, rerunning the reachability analysis using traditional algorithms can be used for counter-example construction. But in doing so, all advantages of the PastFree[ze] algorithm are lost if the traditional verification run fails. To address this problem, an implementation of the PastFree[ze] algorithm could rely on secondary storage (disk) to dump the states to disk (along with links to their parents) before freeing them from memory. In this case, a "fast" run of the PastFree[ze] can be used to identify the failing scenarios followed by a slower execution which relies on the disk-dump for building the counter-example. The advantage of this strategy is twofold: (a), numerous large verification runs can be executed without paying the overhead of secondary storage, (b) in the case of failure, a counter-example can be constructed by simply replaying the failing analysis in the "slow"-mode.

2.2.2 Automated context partitioning

The use of the CDL contexts not only enables the development of new reachability algorithms, but also helps to automatically partition the state space if a given analysis fails due to insufficient memory. Relying on the acyclic nature of the context scenarios, a powerful state-space decomposition technique was introduced by Dhaussy et al. in [18]. This section briefly presents this technique, emphasizing on its complementarity with the PastFree[ze] algorithm.

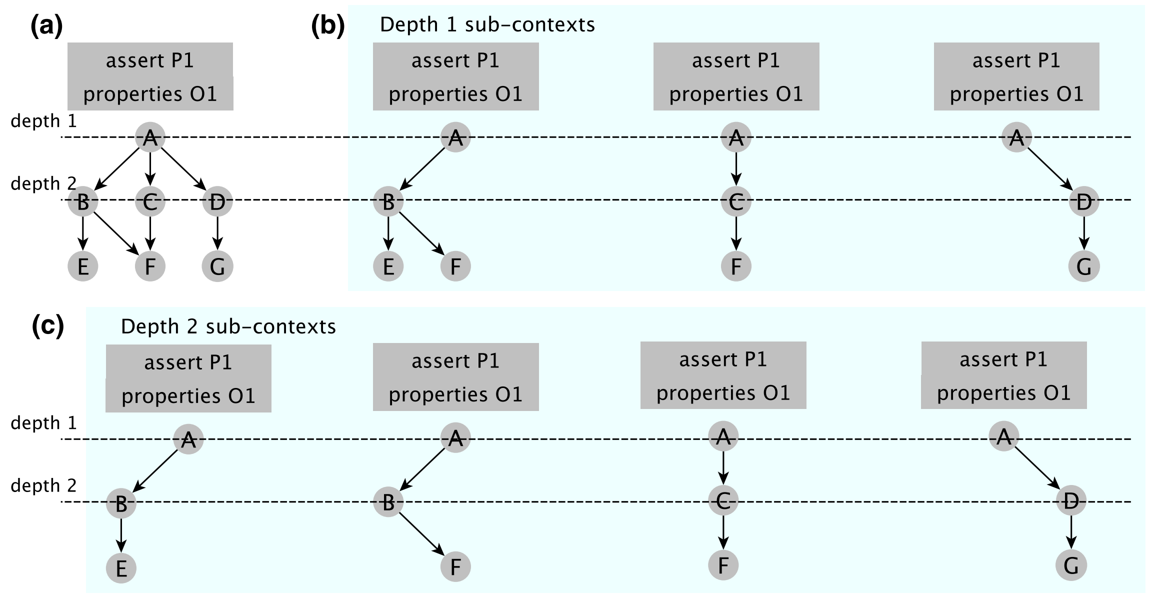

Fig. 2 Illustration of the context splitting algorithm. **a** Verification unit composed of the acyclic interaction scenario T_a and the associated predicate and observers; **b** Automatic split at *depth* 1 results in 3 sub-contexts; **c** Automatic split at *depth* 2 results in 4 sub-contexts; the *gray rectangles* (assert/properties) show the predicates and the observers that are associated with the top-level verification context (**a**). After the split step, these same predicates and properties will be verified for each resulting sub-context

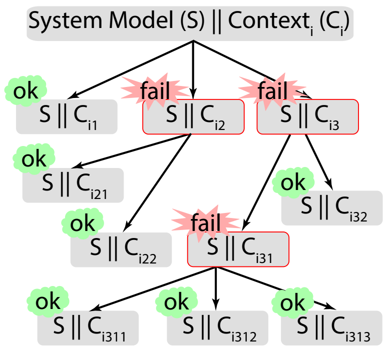

Fig. 3 Split-tree representation of the recursive context partitioning of reachability analysis for $S||C_i$

This state-space decomposition strategy relies on the automated recursive partitioning (splitting) of a given context in independent sub-contexts. A "depth" factor controls the execution of the split in terms of the depth of the interaction scenario. A depth factor of 1 corresponds to the first DAG layer with an alternative (non-deterministic choice). This "depth" factor increases by one for each DAG layer after the first layer with an alternative node. The results of the split algorithm with different depth factors are schematically presented in Fig. 2. This technique is systematically applied when a given reachability analysis ($S||C_i$ in Fig. 3) fails due to the state-space explosion problem. At the beginning of the exploration, the verification engineer can use the "depth" factor to tune the splitting algorithm (by default a depth of one is assumed). After splitting context$_i$, the sub-contexts are iteratively composed with the model for exploration, and the properties associated with context$_i$ are checked for all sub-contexts. Therefore, the global verification problem for context$_i$ is effectively decomposed into K_i smaller verification problems. Hence, verifying the properties on all these K_i problems is equivalent to verifying them on the initial system. In the worst case scenario, the recursive application of the splitting algorithms renders a full split-tree with the leaves being sequential (not branching) interaction scenarios that cannot be decomposed anymore. However, a sequential interaction scenario has the ideal topology for the PastFree[ze] algorithm (maximum two adjacent clusters are present in memory at any time).

As opposed to the PastFree[ze] algorithm that by exploiting the context acyclicity reduces the stress on memory during reachability, the use of the 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 which can slow the verification process. However, in practice this tradeoff is worthwhile since this automated partitioning technique enables the analysis of large problems without the need to buy exponentially more memory. Moreover, due to the independent nature of the automatically generated sub-contexts, this problem can be partially addressed by distributing the verification over a network of computers.

2.3 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*, which implements the PastFree[ze] and the context partitioning algorithms, besides the traditional reachability algorithms (based on breadth/depth first search). Moreover, to foster the generality of CaV and its complementarity with existing model checkers the toolkit provides a bridge [15] to the time petri net analyzer (Tina) [5]. This shows the possibility of benefiting from the methodological advantages of the CaV approach in conjunction with off-the-shelf verification tools. In this case, the recursive splitting algorithm can be used to decompose the interaction scenario before its mapping to the input language of the external tool (Fiacre, in the case of Tina). The use of the PastFree[ze] algorithm is more difficult mainly due to the loss of the context reification in the external tool. Nevertheless, the integration of the PastFree[ze] is possible at the algorithmic level, and can be configured to use automatically detected acyclic components, such as the CDL contexts or other acyclic behaviors of the system.

Figure 4 shows a global overview of the OBP *Observation Engine*. The *System model* representing the SUS is described using the formal language Fiacre [22], which enables the specification of interacting behaviors and timing constraints through a timed-automata-based approach. The environment is decomposed by the designer in one or more verification units expressed with the CDL language. The OBP *Observation Engine*verifies the given set of properties with a reachability strategy on the implicit 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, thus effectively performing an exhaustive state-space analysis. 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 Labelled Transition System (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, hence effectively guiding the user through the process of the SUS evaluation against the given requirements.

Besides Fiacre, the OBP toolkit enables the analysis of SUS models expressed in a restricted, well-formed subset of the UML language [28,29]. Furthermore, the built-in integration of the CCSL [2,30] and MoCCML [14,35] logical-time formalisms paves the way to the integration of formal analysis tools in industrial-scale real-time system engineering methodologies.

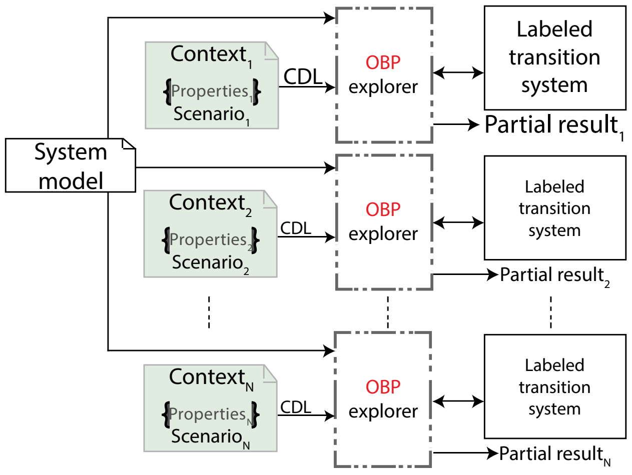

Fig. 4 Context-aware model checking

3 Case study: the landing gear system

In this section, we apply our context-aware verification approach to the LGS case study, we overview the LGS modelling using the Fiacre language, the environment specification using CDL, and we introduce two properties which should be verified by the system.

3.1 Modelling the SUS

The Fiacre LGS model, presented in Fig. 5, 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 failures in the physical components (Fig. 6). The interactions from the environment (i.e. *handle* and *failures*) are managed by a specific component called *Dispatcher*. Inputs are received through an 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.

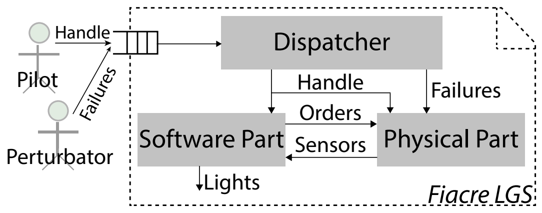

Fig. 5 Overview of the LGS components and the SUS perimeter

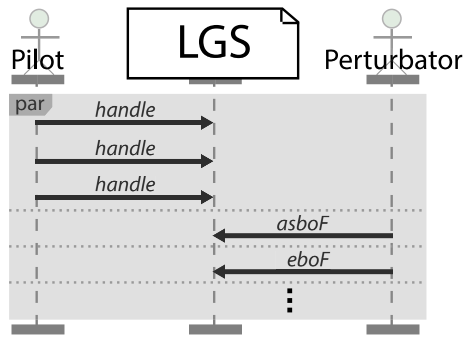

Fig. 6 SUS/environment interaction scenario (note that the Pilot and the Perturbator events are interleaved)

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 1 (top) shows the number of states of each of these processes along with the number of times each one is instantiated in the model.

Similarly, the software part 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. Table 1 (bottom) shows the number of states of each of these processes along with the number of times each one is instantiated in the model.

The Fiacre model of the LGS described in the previous paragraphs has around 3000 lines of code, and it is available at http://www.obpcdl.org along with the OBP *Observation Engine*toolset.

Assumptions and restrictions With respect to the general description of the case study, two more restrictions have been introduced:

1. Firstly, we consider only one software module (and not two as required in the general description), which is assumed failure free.
2. Secondly, we consider only one failure-free wire for each sensor (and not three as required in the general description). Put differently, we suppose that sensors are safe, i.e. without any failure mode. Nevertheless, we assume that all the physical equipment can fail at anytime. However, failures are assumed to be permanent, such that if a equipment (a gear for instance) becomes blocked, then it remains blocked forever.

Except for these restrictions, all the requirements have been taken into account. Particularly, the timing constraints: the automata of the gears, doors, electro-valves, and analog-switch implement the continuous-time behavior as required in the general description. Similarly, the *EV-manager* allows the pilot to change the sequence (from retraction to extension or vice-versa) at anytime during the sequence. Finally, *EV-manager* monitors the physical equipment through the electrical values returned by the sensors. Whenever one of these values is not equal to the one expected by the software part (for instance the right door is still seen closed 7 seconds after activation of the opening electro-valve), then an *anomaly* state is reached and the red light is turned on.

3.2 Modelling the context

As mentioned in the previous section, the environment of the LGS 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. Using the CDL formalism, the pilot behavior is represented through an activity composed of a sequence of N *handle* events sent to the *Dispatcher* process (see first three lines of Listing 3).

The *Perturbator* actor encodes all considered failure configurations composed of sequences of 1 up to 3 failures

Table 1 Fiacre processes for the physical part (top) and software part (bottom)

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

Table 2 Overview of the considered failures along with the affected components

Analog switch		General EV		Door electro-valves				Gear electro-valves			
				Extension		Retraction		Extension		Retraction	
Opened	Closed	Opened	Closed	Opened	Closed	Opened	Closed	Opened	Closed	Opened	Closed
asboF	*asbcF*	*gboF*	*gbcF*	*deboF*	*debcF*	*drboF*	*drbcF*	*geboF*	*gebcF*	*grboF*	*grbcF*
Exclusive		Exclusive		Exclusive		Exclusive		Exclusive		Exclusive	

Door			Gear		
Front	Left	Right	Front	Left	Right
fdF	*ldF*	*rdF*	*fgF*	*lgF*	*rgF*

taken from the total 18 failures that have been identified, see Table 2 for the complete list of the failures classified according to the affected component. 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: (a) 18 possible configurations with 1 failure (the kth failure, in Listing 3, identifies each failure in Table 2 in the left-to-right order—asboF = 1st failure, asbcF = 2nd failure, etc.). (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 is encoded as a CDL activity (Listing 3 lines 7–14), named FailureContext $_k^x$, where $x \in [1 \ldots 3]$ 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. $k \in [1 \ldots 147]$, $for\ x = 2$). 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 3.

The CDL specification of the global environment, Listing 3 lines 20–21, consists of the initialization of the SUS (line 20) followed by the asynchronous interleaving of the $Pilot$ events with the $Perturbator$ failure sequences. Note also the association of the properties to be verified (described in the following paragraphs) with the context (line 18). This verification unit, capturing the complete environment of the LGS, serves as starting point of the verification. However, due to the complexity of the system (leading to state-space explosion) the results, in Sect. 4, are presented with respect to different sub-contexts of this CDL.

Listing 3 Overview of the CDL environment description

```
1  event Handle is  {
2      send HANDLE to {Dispatcher}1}
3  activity PILOT is { loop N event Handle }
4
5  event asboF is {
6      send ASBO_FAILURE to {Dispatcher}1}
7  activity FailureContext$_k^1$ is {
8      event $k^{th}$failure }
9  activity FailureContext$_k^{2..3}$ is {
10 ...// all permutations of $k^{th}$ 2(or 3) failures}
11 activity Perturbator is {
12     FailureContext$_1^1$ [] ··· [] FailureContext$_{18}^1$
13  [] FailureContext$_1^2$ [] ··· [] FailureContext$_{147}^2$
14  [] FailureContext$_1^3$ [] ··· [] FailureContext$_{720}^3$}
15
16 cdl scenario_885_failure_configurations is {
17 // reference to the observers for $R_1$, and $R_2$
18     properties oR$_1$, oR$_2$
19 // environment model
20     init is { act_init }
21     main is { PILOT || Perturbator }
22 }
```

3.3 Specifying the properties

To illustrate the property specification aspects of the CDL language, let us consider the following two requirements:

– *Requirement* R_1 The red light should always be off.
– *Requirement* R_2 At the end of each $Pilot$ interaction, the green light should be on.

Listing 4 CDL-based property specification

```
1  predicate pRed is { {SYS}1:red_light=true }
2  event evt_red is { pRed becomes true }
3  ...
4  property oR$_1$ is {
5      start – evt_red –> reject }
6  property oR$_2$ is {
7      clock ck;
8      start    — evt_orange   / ck := 0
9          –> maneuvering;
10     maneuvering — evt_green / ck := –1
11         –> start;
12     maneuvering — ck>=15000 / ck := –1
13         –> reject;
14 }
```

In CDL, R_1 is an observer that reaches the $reject$ state when the red_light turns on, line 4 in Listing 4. The $\{SYS\}1$

prefix indicates the Fiacre component where the red_light variable is defined. The event evt_red expresses the rising edge of the predicate $pRed$, which refers to the red_light variable defined in the SUS. Similarly, evt_orange and evt_green capture, respectively, the rising edge of the $orange\_light$ and $green\_light$ variables (defined in the LGS specification [7]). The second requirement, R_2, is represented using an observer automaton that follows the system execution and reaches the $reject$ state whenever the green light is not turned on before the ck deadline. The observer declaration (line 6) is introduced with the *property* keyword and defines a transition from the **start** state to the **maneuvering** state initializing the timer ck when the evt_orange is present, a transition from the **maneuvering** state back to the **start** state (disabling the timer), and a transition from the **maneuvering** state to the **reject** state if the timer expired. These observers are referenced in the contexts in which they should be checked and composed with the system during reachability analysis.

4 Landing-gear system: reachability results

This section presents some experimental results obtained using our context-aware verification approach [16] on the LGS. We show the reachability results obtained, emphasizing the importance of the explicit-environment modelling and its use for the methodical decomposition and analysis of large state spaces.

Section 4.1 overviews some specific, practical aspects of the experiments. After introducing the differences with the preliminary results (Sect. 4.2), presented in [19], in Sect. 4.3 we discuss the analysis of the LGS in the presence of one failure. Section 4.4 details the reachability results obtained for two and three LGS failures, overviews some of the advantages of the PastFree[ze] reachability algorithm and discusses some of the limitations of our approach. At last, Sect. 4.5 shows the importance of clearly identified environment models through the analysis of some of the possible alternatives, such as infinite and arbitrary number of pilot interactions with and without failures.

4.1 Practical remarks

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, 64 and 128 GB of available memory.

While the environment model presented in Sect. 3.2 considers only a single top-level environment, our explicit-environment modelling approach also enables the analysis of partial system behavior. For instance, by decomposing the $Perturbator$ actor in multiple partitions, we obtain the set of simpler interaction scenarios featuring sub-sets of all the possible failures.

To ease the analysis and the presentation of the results, the first two levels of decomposition were done manually. Firstly, decomposing the CDL (in Listing 3) in three sub-contexts relative to the number of failures injected (1-failure, 2-failures, 3-failures). Secondly, each one of these contexts was again decomposed according to the types of failures. In this case, the use of the automated splitting algorithm, presented in Sect. 2.2.2, would have provided similar results but with sub-contexts more difficult to present succintly (mainly due to the split-points generated by the interleaving with the Pilot actor).

4.2 Preliminary results

In the preliminary study [19], published at the ABZ 2014 conference, the context-aware verification approach was used to study the LGS. While different from the numerical results in [19], the results presented here do not contradict those initial results, but rather reinforce them. The reasons behind these differences are mainly rooted in the algorithmic improvements integrated in the newer versions of the OBP tools, which now enable the analysis of larger timed systems. Two of the most important improvements are:

- The implementation of an highly optimized time representation, which reduces the memory size of each configuration in the state space;
- The integration of the PastFree[ze] context-driven reachability algorithm, which relying on the explicit-environment representation enables the analysis of larger state spaces;

Besides these core algorithmic improvements, the LGS model itself was simplified, without changing its behavior, by eliminating some of the intermediate states through the use of broadcast channels (supported by Fiacre language) instead of multiple 2-point channels. Moreover, we have eliminated a modelling artefact in the Dispatcher process, which was introducing redundant configuration states in the coupling between the CDL environment and Fiacre system.

Due to these improvements, in this study we will exclusively focus our attention on the results obtained for a 3 gear system, without having to resort to the 1-gear/2-gears partitioning of the Fiacre model used in the previous study [19].

Before diving into a detailed analysis of the reachability results obtained, it is important to note the quality of the LGS case study that captures the complexity and asymmetries of realistic timed systems. The results presented in this section will emphasize the practical advantages of an environment-driven approach to formal verification. The uniqueness of this approach resides on the use of the environment-model

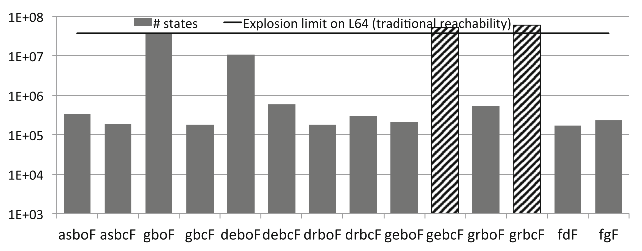

Fig. 7 Reachability results by failure type for 3 Pilot interactions. The *left* (right) door (gear) exploration contexts are not shown since the results are similar to the front door (gear) cases (fdF, fgF). The cases that extend beyond the explosion limit line failed on $L64$. The *bars with the diagonal pattern* show the results obtained on $L128$

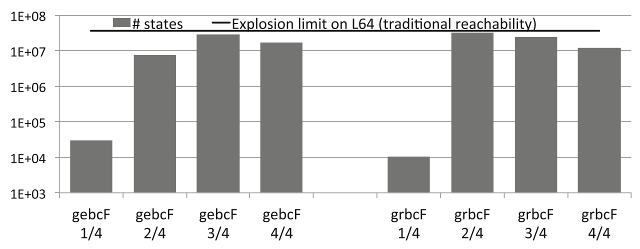

Fig. 8 Reachability results on $L64$ for the 4 partitions obtained for the gebcF and grbcF failures

to guide the verification algorithms in situations otherwise impossible to analyse due to the state-space explosion problem. Moreover, through the isolation and the reification of the environment, our approach offers the basis for the methodical study of its impact on the studied model.

4.3 The impact of the context-aware partitioning: one failure case

In this section, we consider the verification of the full LGS system in the case of one failure. The environment model, presented in Sect. 3.2, was partitioned so that we obtain 18 verification contexts each one with the perturbator actor injecting one failure. The number of pilot interactions was set to three to permit the verification of a full down/up/down sequence.

Compared to the previous results that failed to analyse 6 of the 18 contexts [19], the improvements integrated in the OBP *Observation Engine*and the LGS model enabled the direct verification of all 18 contexts. Figure 7 shows the size of the state space for these verification contexts. On $L64$ the analysis finished successfully for 16 of the 18 context analysed, failing when the gear extension/retraction electro-valves are blocked in the closed position (gebcF, and grbcF). In these cases, the exploration on $L128$ finished successfully, and these results are presented in the figure (the bars with the black-diagonal pattern). The "explosion limit" line in the figure shows the maximum number of states that were successfully analysed on $L64$ using the traditional reachability algorithm (35 701 272 states). In the gebcF (grbcF) case, the exploration on $L64$ unravelled only 79 % (58 %) of the total state space.[5]

Using our environment-driven state-space decomposition technique (introduced in Sect. 2.2.2), we have obtained 4 smaller exploration contexts for both the gebcF and grbcF cases, which have been independently analysed. Through this decomposition, the complete analysis of these two cases was possible. The results are presented in Fig. 8 showing the same "explosion limit" as in Fig. 7. In this case, one important observation is that the partitions obtained by context-splitting are not disjoint. Hence their analysis unravelled 54,462,931 states for gebcF and 68,332,260 states for grbcF that represent the analysis of a state space 1.06, respectively 1.15, times larger than the exact results presented in Fig. 7 for these cases. Another observation is that the analysis of these partitions on $L64$ was 1.30, respectively 1.44, times longer (in terms of exploration time) than the unpartitioned exploration on $L128$. Nevertheless, we believe that this is a small price to pay for the opportunity of analysing systems (almost) twice larger without the need of doubling the physical memory of the machine.

4.4 Pushing the limits: multiple LGS failures

In the presence of two failures, the analysis of most contexts finishes successfully; however, 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 (for brevity reasons, the results after partitioning are not presented in these cases). Figure 9 (top) shows the reachability results for 64 of the two-failure contexts (asboF, asbcF, gboF, gbcF followed by all other failures). The total number of states explored for these contexts exceeds 2 billion. In this case, we have used the PastFree[ze] reachability algorithm. This algorithm enabled the exploration of a larger state space. The explosion limit line on $L64$ was pushed from 35,701,272 states (in Fig. 7) to 69,553,139 states (in Fig. 9), representing the exploration of a state space almost twice larger (1.94 times larger).

As stated in Sect. 2.2.1, the PastFree[ze] algorithm relies on the acyclic environment specifications to prune the state space of clusters with past configurations (with respect to the environment). To grasp the importance of this approach, Fig. 9 (bottom) shows the percentage of the state space that

[5] It should be noted that due to the use of a "log"-axis in Fig. 7 it is difficult to see that the state spaces for these two cases exceed the $L64$ threshold by 21 and 41 %, respectively.

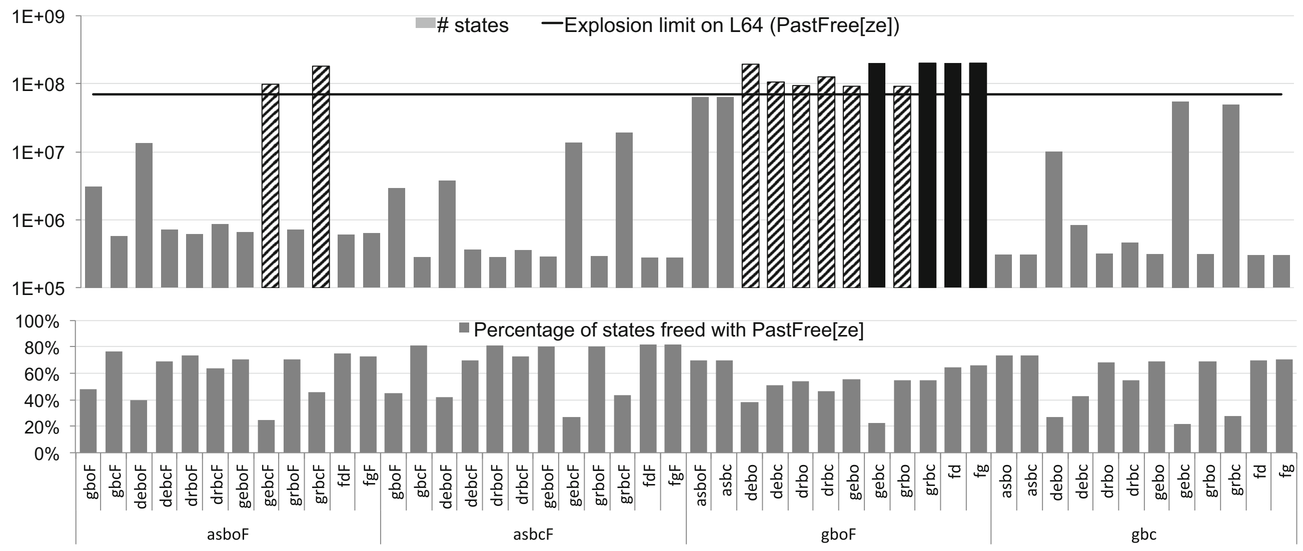

Fig. 9 *Top* reachability results for 3 Pilot interactions interleaved with some of the 2 failure combinations. The cases that extend over the explosion limit failed on $L64$. The *black bars* show the cases where the analysis failed on $L128$. *Bottom* the percentage of the state-space freed from memory using the PastFree[ze] reachability

was freed from memory during the analysis of each context. 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 of the 64 contexts presented in Fig. 9.

In the presence of three failures, the complexity of the LGS model emphasizes the space/time trade-off of our approach. Partitioning the state space in independent verification runs renders the exhaustive verification of each partition manageable (in terms of space). However, the larger the state space, the larger is the number of partitions to verify. In the case of the LGS, there are 720 3-failure contexts that have to be explored. Supposing that these verification runs are performed on 16 independent computers, and that each run takes 1 h to finish that means the whole verification will take 45 h. To cope with this difficulty, we have used the alternative operator in the CDL (non-deterministic choice) to group multiple 3-failure contexts together. The types of failures considered in these composite contexts excluded the ones that generated more than one million states in the one-failure case. This choice rendered the verification contexts manageable in terms of the size of the state space. Figure 10 shows the results for three such composite contexts. The perturbator actor in the first case ($s1$) injects: (asboF [] asbcF[] gbcF) **;** (drboF [] drbcF [] geboF) **;** (fdF [] fgF [] debcF), where the [] and **;** are the non-deterministic choice, and the sequential operators in the CDL language. The perturbator in the $s2$ and $s3$ cases is the two permutations of the $s1$ on the sequence operator (**;**). In this way, 81 of the 720 contexts were explored in three verification runs that took 5 h to finish successfully. The "# states freed" part of the bars in Fig. 10 shows the number of

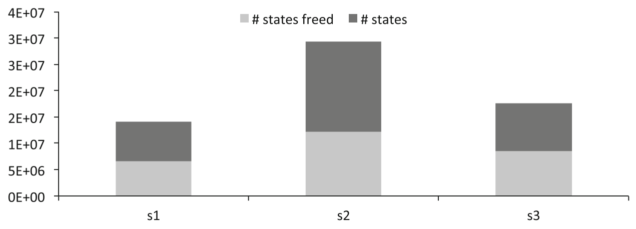

Fig. 10 Reachability results for 3 composite contexts integrating 81 primitive 3-failure contexts

states that were freed when using the PastFree[ze] algorithm for the exploration, which amounts to 45.5 % on average.

For the purpose of this study, it is important to note that, while the number of primitive partitions increases exponentially in the case of "three failures", the CDL language offers the possibility to merge the primitive partitions. By merging these partitions, with the CDL parallel composition operator, larger verification contexts are created, and the advantages of reified environment specifications are maintained.

4.5 Cyclic and arbitrarily long contexts

In the last sections, we have presented the reachability results of the LGS in the presence of failures restricting to three the number of Pilot interactions in the environment model. This section investigates the impact of infinite and arbitrary number of pilot interactions on the LGS reachability analysis in the nominal mode and one-failure mode.

Infinite pilot interactions To facilitate the implementation of context-driven verification techniques (such as the automatic-split and the PastFree[ze] reachability), for now

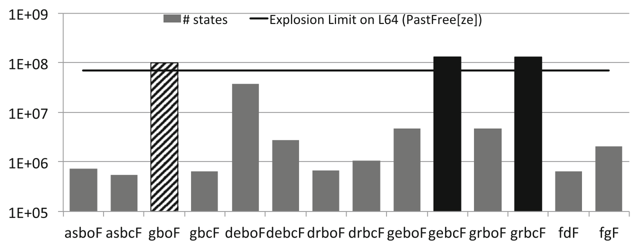

Fig. 11 Reachability results for Infinite Handle interactions for the one-failure contexts. The *bars* with the diagonal pattern failed on L64 and show the results on L128. The *black bars* failed on both L64 and L128

the CDL language does not permit the expression of cyclic environment models. To model an infinite number of pilot interactions, the pilot actor was removed from the CDL specification and was included in the Fiacre model as a one-state process with a single transition that sends the Handle command to the Dispatcher process (without timing restrictions between the occurrence of two Handle commands). In the nominal mode (no failures present), the reachability analysis finishes successfully on $L64$ unravelling 352,379 states and 1,477,197 transitions in 226.9 s. However, when considering the injection of one failure in the model the impact on the feasibility of the verification is very important. Figure 11 overviews the results for 14 of the 18 failure cases.[6] In most cases, the reachability analysis finishes successfully on $L64$ in the presence of one failure (the gray bars in the figure) but it fails on $L64$ for the gboF, gebcF and grbcF. More importantly, if the gear electro-valves block in the closed position (gebcF and grbcF) the exploration also fails on the $L128$ (the black bars in the figure). In these two cases, due to the introduction of the Pilot actor in the Fiacre model, neither our state-space decomposition nor the PastFree[ze] algorithm helps and typically the number of pilot interactions and/or the occurrences of the failure are restricted in an model-dependent ad-hoc manner.

Arbitrary number of pilot interactions To cope with the state-space explosion in the previous case, we could restrict the number of pilot interactions, as we did in the previous sections. 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 is 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 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 the environment becomes cyclic. Table 3 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 $n \in [3 \ldots 498]$), which holds for the next 994 handles. This pattern provides a strong indication of the cyclicality of the system modulo the environment model, which can be proved by bisimulation on the two state spaces induced by the environment.

To show the scalability of our approach, Fig. 12 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 3) lower 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 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 sent 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 traditional reachability analysis all cases with more than 400 handles fail on L64 due to the lack of memory. Figure 13 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 80 GB (87,540,904 states). Using our context-driven reachability algorithm, the analysis of all 1000 cases was successful using less than 10 GB of RAM, which represents only 15 % of the total amount of memory available on $L64$.

To conclude, we have to acknowledge the complexity of the LGS model, which challenged our methodology and tools, and emphasized once more the exponential growth in complexity that fuelled decades of research in formal verification. The context-aware verification approach introduces a new axis for taming the state-space explosion problem [39], which is complementary with more holistic approaches such as partial-order reduction [38], and symmetry reduction [12].

5 Related work

Since the introduction of model checking in the early 1980s [33], several model-checker tools have been developed to help the verification of concurrent systems [4,26,42].

[6] the left/right door/gear failure cases are not include since they produce the same results as the front door/gear—ldF, lgF.

Table 3 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	1326	1	448	12,904	44,331	72,650	85,042	87,780	87,746	87,780	87,746

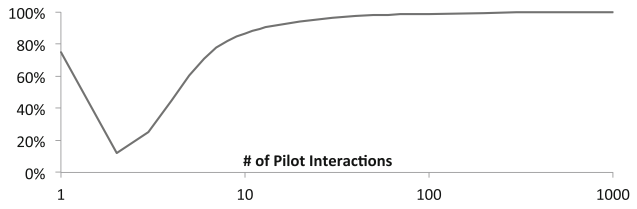

Fig. 12 Percentage of the state-space freed due to PastFree[ze]—1 to 1000 handle occurrences

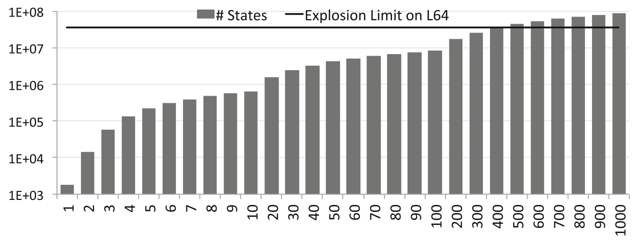

Fig. 13 Reachability results for 1 to 1000 pilot interactions in the nominal mode

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

Some researchers propose to prune the state space using techniques such as partial-order reduction [24,32,38] and symmetry reduction [12] 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.

The use of efficient data structures, such as Binary Decision Diagrams (BDD) [9], for achieving compact state-space representation gave rise to a whole class of model-checking tools, typically known as symbolic model checking. The OBP *Observation Engine*toolkit, however, uses a different approach, known as explicit-state model checking. Instead of using BDDs, the state space is explicitly stored using dictionary-like data structures. This approach facilitates the use of external storage [34], and eases the decomposition of the state space during reachability, a feature needed for freeing the clusters of states corresponding to past contexts steps. For reducing the memory requirements, in this setting, OBP *Observation Engine*splits each state into its components (eg. variables), which are shared between different states. This strategy is similar to the recursive indexing method introduced by Holzmann [25] (the COLLAPSE method in the Spin model checker).

To reduce impact of the state-space explosion problem, other approaches, such as Murpφ [34], TLC [21] or semi-external LTL model checking [42], focus on algorithmic advancements and the maximal use of the available resources such as external memories (disk). The PastFree[ze] algorithm belongs to this class of strategies, since it exploits the context information to reduce the memory requirements during reachability. Moreover, if the generation of a counter-example is needed, the PastFree[ze] algorithm becomes a semi-external algorithm (the data structure is distributed in memory and on disk), storing on disk the clusters freed from memory. However, as opposed to the previously cited approaches, our algorithm exploits the structural characteristics of the system to minimize the number of IO operations (no need to read previously saved configurations during the analysis).

Techniques such as bounded model checking [10] (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 BMC. The usage of acyclic behaviors offers more flexibility for specifying the “bounds” of the analysis, and the interaction scenarios can be seen as a high-level skeleton which drives the analysis through a complex state-space partition. We are currently investigating the integration of BMC with CaV approach. An acyclic CaV context can be obtained by performing BMC on the “environment” of the SUS. In this case, through BMC, arbitrary contexts (cyclic) could be transformed to “bounded” acyclic verification units which can then be fully exploited using the CaV approach. From a methodological point of view, such an integration presents good methodological properties due to the fact that the SUS can be exhaustively analysed (no bounds on the SUS behaviors) and that the coverage proofs are restricted only to the environment component.

While the previous techniques address the property verification problem monolithically, compositional verification [23] 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 approach used for the study of the LGS can be seen as coarse-grain compositional verification, where the focus is steered towards the interactions of the whole system with its surrounding environment (context). Nevertheless, in conjunction with the right composition operators, the CaV method can also be used for component-level verifications.

Conversely to "traditional" techniques in which the surrounding environment is often implicitly modeled in the system (to obtain a closed model), a number of techniques have been proposed for explicitly capturing the environment characteristics in isolation from the model [37,41]. While similar to these works, the Context-aware Verification approach, presented in this study, goes one step further by reifying the environment instead of using this description only for code generation. Practically, this enables the implementation of context-aware algorithms, such as PastFree[ze] and automated context partitioning, reducing the impact of the state-space explosion.

6 Conclusion and perspectives

In this paper, we apply the Context-aware Verification technique to the Landing Gear System. This approach based on Fiacre and CDL languages closes the system-under-study with a well-defined environment. For the LGS, a top-level context was used to capture the requirements and the interaction scenarios with the environment. The decomposition of this context produced 885 isolated smaller verification units. The analysis of 18 % of these unravelled a state space of over 2.2 billion states. The positive results shown in this paper rely on the algorithmic exploitation of the interaction scenarios between the system-under-study and its environment. A context-driven reachability algorithm was presented, which reduces the memory consumption during state-space exploration, and enables the analysis of the LGS with an arbitrary number of Pilot interactions. The impact of the state-space explosion problem is further reduced by the automated recursive partitioning of the contexts.

Besides the integration with BMC, briefly mentioned in Sect. 5, future research directions include the design of a higher-level environment specification formalism. This will ease the specification of large interaction scenarios while preserving the virtues of the CDL language. Moreover, for the effective industrialisation of this verification approach, a clear methodological tool-supported framework should be developed. This framework has to: (a) be seamlessly integrated with industry practices, (b) to provide coverage metrics reflecting the quality of the environment model, (c) to support the distributed execution of a large number of verification units, and (d) to offer scalable tools for counter-example-based diagnosis.

Acknowledgments The authors wish to thank Dr. Frederic Boniol for his valuable and constructive suggestions related to this paper.

References

1. Alur, R., Dill, D.L.: A theory of timed automata. Theor. Comput. Sci. **126**, 183–235 (1994)
2. André, C.: Syntax and Semantics of the Clock Constraint Specification Language (CCSL). Research Report RR-6925, INRIA, (2009)
3. Armbrust, M., Fox, A., Griffith, R., Joseph, A.D., Katz, R.H., Konwinski, A., Lee, G., Patterson, D.A., Rabkin, A., Stoica, I., Zaharia, M.: Above the clouds: a berkeley view of cloud computing. Technical Report UCB/EECS-2009-28, EECS Department, University of California, Berkeley (2009)
4. Bengtsson, J., Larsen, K.G., Larsson, F., Pettersson, P., Yi, W.: UPPAAL—a tool suite for automatic verification of real–time systems. In: Proceedings of Workshop on Verification and Control of Hybrid Systems III, number 1066 in Lecture Notes in Computer Science, pp. 232–243. Springer–Verlag (1995)
5. Berthomieu, B., Ribet, P.-O., Verdanat, F.: The tool TINA—construction of abstract state spaces for petri nets and time petri nets. Int. J. Prod. Res. **42**, 2741–2756 (2004)
6. Boniol, F., Dhaussy, P., Le Roux, L., Roger, J.-C.: Model-Based Analysis. In: Embedded Systems, pp. 157–183. Wiley, New Jersey (2013). doi:10.1002/9781118569535.ch8
7. Boniol, F., Wiels, V.: The landing gear system case study. In: Boniol, F., Wiels, V., Ait Ameur, Y., Schewe, K.-D. (eds.) ABZ 2014: The Landing Gear Case Study, volume 433 of Communications in Computer and Information Science, pp. 1–18. Springer International Publishing (2014)
8. Boniol, F., Wiels, V., Ledinot, E.: Experiences using model checking to verify real time properties of a landing gear control system. France, In: Embedded Real-Time Systems (ERTS), Toulouse (2006)
9. Burch, J.R., Clarke, E.M., McMillan, K.L., Dill, D.L., Hwang, L.J.: Symbolic model checking: 10^{20} states and beyond. In: 5th IEEE Symposium on Logic in Computer Science, pp. 428–439 (1990)
10. Clarke, E., Biere, A., Raimi, R., Zhu, Y.: Bounded model checking using satisfiability solving. Form. Methods Syst. Des. **19**(1), 7–34 (2001)
11. Clarke, E.M., Emerson, E.A., Sistla, A.P.: Automatic verification of finite-state concurrent systems using temporal logic specifications. ACM Trans. Program. Lang. Syst. **8**(2), 244–263 (1986)
12. Clarke, E.M., Enders, R., Filkorn, T., Jha, S.: Exploiting symmetry in temporal logic model checking. Form. Methods Syst. Des. **9**(1–2), 77–104 (1996)
13. Cormen, T.H., Leiserson, C.E., Rivest, R.L., Stein, C.: Introduction to Algorithms, 3rd edn. The MIT Press (2009)
14. Deantoni, J., Issa Diallo, P., Teodorov, C., Champeau, J., Combemale, B.: Towards a Meta-Language for the Concurrency Concern in DSLs. In: Design, Automation and Test in Europe Conference and Exhibition (DATE), Grenoble, France (2015)
15. Dhaussy, P., Boniol, F., Roger, J.-C.: Reducing state explosion with context modeling for model-checking. In: 13th IEEE International High Assurance Systems Engineering Symposium (Hase'11), Boca Raton, USA (2011)
16. Dhaussy, P., Boniol, F., Roger, J.-C., Le Roux, L.: Improving model checking with context modelling. Advances in Software Engineering, ID 547157:13 pages (2012)

17. Dhaussy, P., Pillain, P.-Y., Creff, S., Raji, A., Le Traon, Y., Baudry, B.: Evaluating context descriptions and property definition patterns for software formal validation. In: Schuerr, Bran Selic Andy (ed.) 12th IEEE/ACM conf Model Driven Engineering Languages and Systems (Models'09), vol. 5795, pp. 438–452. Springer-Verlag, LNCS (2009)
18. Dhaussy, P., Roger, J.-C., Boniol, F.: Context aware model-checking for embedded software. In: Embedded Systems—Theory and Design Methodology, pages ISBN: 978–953–51–0167–3 pages 167–184. InTech (2012)
19. Dhaussy, P., Ciprian, T.: Context-aware verification of a landing gear system. In: Boniol, F., Wiels, V., Ait Ameur, Y., Schewe, K.-D. (eds.) ABZ 2014: The Landing Gear Case Study volume 433 of Communications in Computer and Information Science, pp. 52–65. Springer International Publishing (2014)
20. Dwyer, M.B., Avrunin, G.S., Corbett, J.C.: Property specification patterns for finite-state verification. In: FMSP, pages 7–15 (1998)
21. Edelkamp, S., Sanders, P., Šimeček, P.: Semi-external ltl model checking. In: Gupta, A., Malik, S. (eds.) Computer Aided Verification, volume 5123 of Lecture Notes in Computer Science, pp. 530–542. Springer, Berlin Heidelberg (2008)
22. Farail, P., Gaufillet, P., Peres, F., Bodeveix, J.-P., Filali, M., Berthomieu, B., Rodrigo, S., Vernadat, F., Garavel, H., Lang, F.: Fiacre: an intermediate language for model verification in the TOPCASED environment. In: European Congress on Embedded Real-Time Software (ERTS), Toulouse. SEE (2008)
23. Flanagan, C., Qadeer, S.: Thread-modular model checking. In: SPIN'03 (2003)
24. Godefroid, P.: The Ulg partial-order package for SPIN. SPIN Workshop (1995)
25. Holzmann, G.J.: State compression in SPIN: Recursive indexing and compression training runs. In: Proceedings of 3rd International SPIN Workshop (1997)
26. Holzmann, G.J.: The model checker SPIN. Softw. Eng. **23**(5), 279–295 (1997)
27. INCOSE: INCOSE Systems Engineering Handbook: A Guide for System Life Cycle Processes and Activities. Wiley (2015)
28. Jouault, F., Delatour, J.: Towards fixing sketchy UML models by leveraging textual notations: Application to real-time embedded systems. In: Proceedings of the 14th International Workshop on OCL and Textual Modelling co-located with 17th International Conference on Model Driven Engineering Languages and Systems (MODELS 2014), Valencia, Spain, September 30, 2014., pages 73–82 (2014)
29. Jouault, F., Teodorov, C., Delatour, J., Le Roux, L., Dhaussy, P.: Transformation de modèles UML vers Fiacre, via les langages intermédiaires tUML et ABCD. Génie logiciel, 109:xx (2014)
30. Menad, N., Dhaussy, P.: A transformation approach for multiform time requirements. In: Hierons, R.M., Merayo, M.G., Bravetti, M. (eds.) Software Engineering and Formal Methods, volume 8137 of Lecture Notes in Computer Science, pp. 16–30. Springer Berlin Heidelberg (2013)
31. Park, S., Kwon, G.: 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), vol. 3984, pp. 905–911. Springer-Verlag, LNCS (2006)
32. Peled, D.: 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, Springer-Verlag (1994)
33. Queille, J.-P., Sifakis, J.: Specification and verification of concurrent systems in cesar. In: Proceedings of the 5th Colloquium on International Symposium on Programming, pages 337–351, London, UK, Springer-Verlag (1982)
34. Stern, U., Dill, D.L.: Using magnetic disk instead of main memory in the Murφ verifier. In: Hu, A.J., Vardi, M.Y. (eds.) Computer Aided Verification, volume 1427 of Lecture Notes in Computer Science, pp. 172–183. Springer, Berlin Heidelberg (1998)
35. Teodorov, C.: Embedding multiform time constraints in smalltalk. In: Proceedings of the International Workshop on Smalltalk Technologies, IWST '14 (2014)
36. Teodorov, C., Le Roux, L., Dhaussy, P.: Context-aware verification of a cruise-control system. In: Ait Ameur, Y., Bellatreche, L., Papadopoulos, G.A. (eds.) Model and Data Engineering, volume 8748 of Lecture Notes in Computer Science, pp. 53–64. Springer International Publishing (2014)
37. Tkachuk, O., Dwyer, M.B., Pasareanu, C.S.: Automated environment generation for software model checking. In: Automated Software Engineering, 2003. In: Proceedings of 18th IEEE International Conference on, pages 116–127 (2003)
38. Valmari, A.: 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, Springer-Verlag (1991)
39. Valmari, A.: The state explosion problem. In: Reisig, W., Rozenberg, G. (eds.) Lectures on Petri Nets I: Basic Models, volume 1491 of Lecture Notes in Computer Science, pp. 429–528. Springer, Berlin Heidelberg (1998)
40. Whittle, J.: 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 (2006)
41. Yatake, K., Aoki, T.: Automatic generation of model checking scripts based on environment modeling. In: Proceedings of the 17th International SPIN Conference on Model Checking Software, SPIN'10, pages 58–75, Berlin, Heidelberg, Springer-Verlag (2010)
42. Yu, Y., Manolios, P., Lamport, L.: 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, Springer-Verlag (1999)