Context-Aware Verification of a Landing Gear System

Philippe Dhaussy and Ciprian Teodorov

UEB, Lab-STICC Laboratory UMR CNRS 6285
ENSTA Bretagne, France
{firstname.name}@ensta-bretagne.fr

Abstract. Despite the high level of automation, the practicability of formal verification through model-checking of large models is hindered by the combinatorial explosion problem. In this paper we apply a novel context-aware verification technique to the Landing Gear System Case Study (LGS) [2]. The idea is to express and verify requirements relative to certain environmental situations. The system environment is decomposed into several independent scenarios (contexts), which are successively composed with the system during reachability analysis. These contexts are specified using a language called CDL (*Context Description Language*), based on activity and message sequence diagrams. The properties to be verified are specified with observer automata and attached to specific regions in the context. This approach enables an automated context-guided decomposition of the verification into smaller problems, hence effectively reducing the state-space explosion problem. In the case of the LGS this technique enabled the fully-automated decomposition of the verification into 885 smaller model-checking problems.

Keywords: formal verification, context-aware model-checking, OBP, observer-automata.

1 Introduction

Software verification is an integral part of the software development lifecycle, the goal of which is to ensure that software fully satisfies all the expected requirements. Reactive systems are becoming extremely complex with the huge increase in high technologies. Among reactive systems, the asynchronous systems communicating by exchanging messages via buffer queues are often characterized by a vast number of possible behaviors. To cope with this complexity, manufacturers of industrial systems make significant efforts in testing and simulation to successfully pass the certification process. Nevertheless revealing errors and bugs in this huge number of behaviors remains a very difficult activity. An alternative method is to adopt formal methods, and to use exhaustive and automatic verification tools such as model-checkers.

Model-checking algorithms can be used to verify requirements of a model formally and automatically. However, because of the internal complexity of the

F. Boniol et al. (Eds.): ABZ 2014 Case Study Track, CCIS 433, pp. 52–65, 2014.

developed systems, model-checking can lead to an unmanageable large state-space, a problem known as the state-space explosion problem [6,16]. Numerous techniques, such as symbolic model-checking [4], and partial-order reduction [20], have been proposed to reduce the impact of this problem effectively pushing the limits of model-checking further and further.

In this paper, we use a novel technique, dubbed context-aware verification [10], to model and analyze the Landing Gear System Case Study LGS [2]. This technique proposes to reduce the set of possible behaviors (and thus the state-space) by closing the system-under-study (SUS) with a well defined environment (context). In the context of embedded reactive systems, the environment of each system is finite and well known. Hence, we claim that the explicit and formal specification of this context enables at least three different state-space reduction axes: *a*) the environment can be decomposed in contexts, thus isolating different operating modes; *b*) each individual context can automatically be subdivided in independent verification problems; *c*) the requirements, specified as observer automata, are focused on specific environmental conditions.

For the LGS , we have modeled one top-level context which was automatically decomposed into 885 isolated smaller scenarios, enabling us to iteratively perform reachability analysis on each of them. Even though, some of these scenarios fail due to the state-space explosion problem we show that our context-aware verification approach pushes the limits of reachability analysis, enabling an automatic divide-and-conquer approach to model-checking. Because the limited size of this paper, we briefly present the SUS modeling and requirement specification, we deliberately focus the presentation on our context-guided state-space reduction technique.

Paper Organization. Section 2 presents the related techniques addressing the state-space explosion problem. Section 3 overviews the principles of our approach for context-aware formal verification. The LGS model is presented, in Section 4, along with the results obtained with OBP *Observation Engine*. Section 5 concludes this study giving some future research directions.

2 Related Work

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 [18], several model-checker tools have been developed to help the verification of concurrent systems [15,1].

However, while model-checking provides an automated rigorous framework for formal system validation and verification, and has successfully been applied on industrial systems it suffers from the state-space explosion problem. 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, thus enabling the verification of ever larger systems. Some of these approaches focus on the use of efficient data-structures such as BDD [4] for achieving compact state-space representation, others rely on algorithmic advancements and the maximal use of the available resources such as external memories [11]. To prune the state-space, techniques such as partial-order reduction [14,17,20,14] and symmetry reduction [7] exploit fine-grain transition interleaving symmetries and global system symmetries respectively. Yet other approaches, like bounded model-checking [5] exploit the observation that in many practical settings the property verification can be done with only a partial (bounded) reachability analysis.

The successful application of these methods to several case studies (see for instance [3] for aerospace examples) demonstrates their maturity in the case of synchronous embedded systems. However, even though these techniques push the limits of model-checking ever further, the state-space explosion problem remains especially in the case of large and complex asynchronous systems.

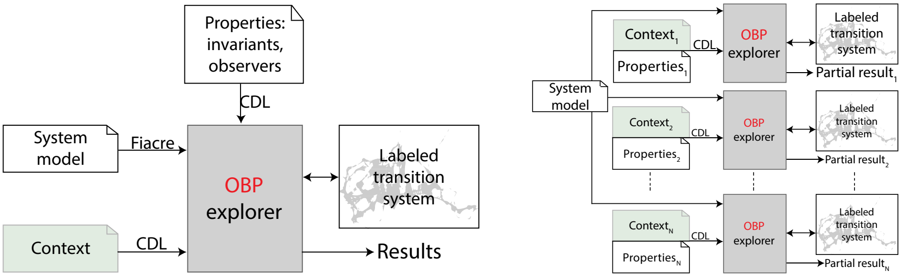
Besides the previously cited techniques that approach the property verification problem monolithically, compositional verification [13] focus on the analysis of individual components of the system using assume/guarantee reasoning (or design-by-contract) to extract (sometimes automatically) the interactions 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.

Our approach can be seen as a coarse-grain compositional verification, where instead of analyzing the interactions of individual components with their neighboring environment we focus on 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), we explicitly describe it separately from the model. By explicitly modeling the environment as one (or more) formally defined context(s) and composing it with the system-under-study we can conduct the full system verification. Using the "context" knowledge the verification problem is decomposed, following a fully automatic divide-and-conquer algorithm, in smaller problems (with smaller state-space) which are analyzed independently.

3 Context-Aware Model-Checking

In this section, we present a formal verification approach that aims primarily at reducing the state-space explosion problem in the context of exhaustive verification through model-checking. This approach, dubbed *context-aware model-checking*, focuses on the explicit modeling of the environment as one or more contexts, which then are iteratively composed with the system-under-study (SUS). The requirements are associated and verified in the contexts that correspond to the environmental conditions in which they should be satisfied, and automated context-guided state-space reduction techniques can be used to further push the limits of reachability analysis. All these developments are implemented in the OBP *Observation Engine*[9] and are publicly available[1].

When verifying properties, through explicit-state model checking, the system explores all the behaviors possible in the SUS and checks whether the verified properties are true or not. Due to the exponential growth of system states relative to the number of interacting components, most of the time the number of reachable configurations is too large to be contained in memory. Besides using techniques like the ones described in Sec. 2, to alleviate this problem the system designers manually tune the SUS 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 SUS should be kept sound, in sync and maintained. To address these issues, we propose to restrict model behavior by composing it with an explicitly defined environment that interacts with the SUS. The environment enables a subset of the behavior of the model. This technique reduces the complexity of the exploration by limiting its scope to a reduced set of behaviors related to specific environmental conditions. Moreover, this approach solves the methodological issues, since it decouples the SUS from its environment, thus allowing their refinement in isolation.



(a) Global view (b) Context decomposition

Fig. 1. Context-aware model-checking

Fig. 1a shows a global overview of the OBP *Observation Engine*. The *System model* representing the SUS is described using the formal language Fiacre [12], which enables the specification of interacting behaviors and timing constraints through a timed-automata based approach. The surrounding environment and the requirements are specified with the *Context Description Language* (CDL). The CDL formalizes the environment through a number of contexts that interact asynchronously with the SUS. Moreover, the CDL enables the specification of requirements through properties that are verified by OBP *Observation Engine*. These properties expressed through property-pattern definitions[10] are based on events (eg. variable x changed) and predicates which are composed to express either invariants or observers. It should be noted that most of the safety properties that we are studying can be expressed using observer automata, moreover in

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

[19] the authors present an automated approach for reducing liveness checking to safety verification by observer-based model instrumentation.

The OBP *Observation Engine*verifies the given set of properties with a reachability strategy using a breath-first-search algorithm on the implicit graph induced by the parallel composition of the SUS with 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 truth values 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 *success/reject* state of a given observer, hence effectively guiding the user through the process of the SUS evaluation against the given requirements.

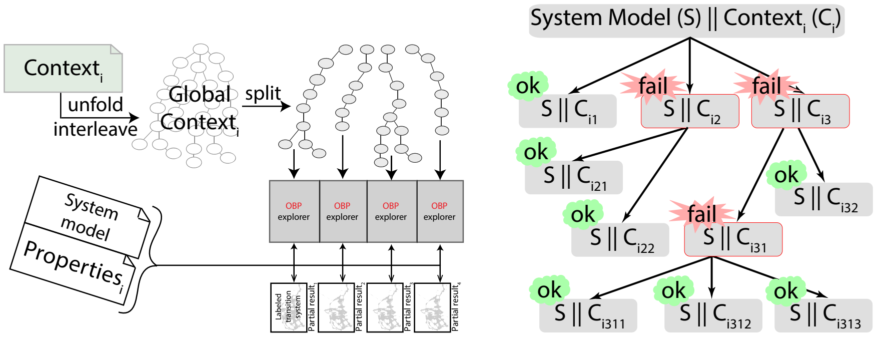
Environment Modeling with CDL Formalism

In the context of reactive embedded systems, the environment of each component of the system is often well known. It is therefore more effective to identify and better express this environment than trying reduce the state-space of the SUS. However, it should be noted that the proof relevance 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 strong hypothesis we argue that 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.* So, we suppose 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 the verification.

To formalize the context specification in [8] we introduced the CDL formal language to capture the interactions with the environment. A CDL[2] model describes the surrounding environment of a SUS and the properties to be checked in this environment. The interleaving of context actors described by a CDL specification generates a graph representing all executions of the environment actors, which can be fed as input to traditional model-checkers, see [8] for more details.

Moreover, if all the identified contexts are finite and acyclic (there are no infinite loops in the interaction between the system and its environment) then the *interleaved context graph* is also finite and acyclic. This is the case with many command systems or communication protocols. Based on this observation we have developed a powerful context-guided state-space reduction technique which relies on the automated recursive partitioning (splitting) of a given context in indepen-

[2] For the detailed syntax, see `www.obpcdl.org`



(a) Acyclic context splitting

(b) Decomposition of reachability analysis for $S||C_i$

Fig. 2. Context-guided state-space reduction and verification

dent sub-contexts. This technique, schematically presented in Fig. 2a, is systematically applied by OBP *Observation Engine*when a given reachability analysis ($S||C_i$ in Fig. 2b) fails due to lack of memory resources to store the state-space. 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.

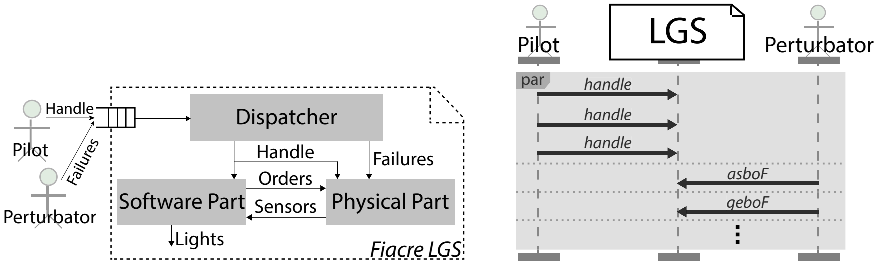
Context-aware reduction of system behavior is particularly interesting in the case of complex embedded system, such as avionics, since they exhibit clearly identified operating modes with specific properties associated with these modes. Unfortunately, only few existing approaches propose practical ways to precisely capture these contexts in order to reduce formal verification complexity and thus improve the scalability of existing model checking approaches. Moreover, a clear methodology that formalizes the context coverage with respect to the full system behavior and assist the user on initial context specification is required for these techniques to be used on industrial-scale critical systems.

4 Case-Study: The Landing Gear System

In this section we apply our context-aware verification approach to the LGS case-study [2] of the ABZ 2014 conference. Before presenting our results, we overview the LGS modeling using the fiacre language, the environment specification using CDL, and we introduce two properties which should be verified on the system.

4.1 Modeling the SUS

The FIACRE LGS model, presented in Fig. 3a, 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 or vice-versa), and a a virtual agent called *Perturbator* injecting failures in the physical components (Fig. 3b). The interactions from the environment (i.e., *handle* and *failures*) 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.



(a) Global view (b) LGS Context Interactions

Fig. 3. Landing gear system model

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

Table 1. Fiacre processes for the Physical Part

	Analog Switch	General_EV	Generic_EV	Gear	Door
# of states	18	34	24	23	20
# of instances	1	1	4	3	3

Each process is a FIACRE automaton. As illustration, Fig. 4 shows the automaton of the process `AnalogSwitch` composed of 18 states. This process implements a loop from *open* to *closed* and from *closed* to *open* through numerous intermediate states including timers as required in the general description of the case study. The two final states at the right of the automaton implements the failure state *blockedOpen* to *blockedClosed*. These states are reached from anywhere in the automaton whenever a failure event is received from the *Perturbator* through the *Dispatcher*.

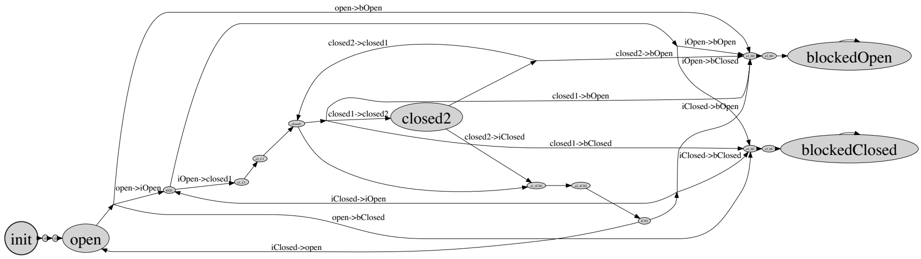


Fig. 4. Automaton of the *Analog Switch* process

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 2 shows the number of states of each of these processes along with the number of times each one is instantiated in the model.

Table 2. Fiacre processes for the Software Part

	Door sensor synth.	Gear sensor synth.	EV Manager	Status Manager
# of states	8	8	52	10
# of instances	3	3	1	1

The FIACRE model of the LGS described in the previous paragraphs has more than 3,000 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 these restrictions, all the other specification have been taken into account. Particularly the timing constraints: the automata of the gears, doors, electro-valves, and analog-switch implement the timed 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.

4.2 Modeling 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 *handle* events sent to the *Dispatcher* process (see first two lines of Listing. 1).

Table 3. 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*

The *Perturbator* actor encodes all considered failure configurations composed of sequences of 1 up to 3 failures taken from the total 18 failures that have been identified, see Table 3 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 follow that there are 885 possible failure configurations as follows: *a*) 18 possible configurations with 1 failures. *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 as encoded as a CDL activity (Listing 1 lines 5–6), 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 configuration to play, see lines 8–11 in Listing 1.

The CDL specification of the global environment, Listing 1 lines 13–16, consists of the initialization of the SUS (line 15) 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 (lines 14).

Listing 1. Overview of the CDL environment description

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

4.3 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 2. 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 { start -- / / evt_red / -> reject }
5  property oR$_2$ is {
6       clock ck;
7       start -- / / evt_orange / ck := 0 -> maneuvering;
8       maneuvering -- / / evt_green / ck := -1 -> success;
9       start -- ck >= 15000 / / / ck := -1 -> reject;
10 }
```

In CDL, R_1 is an observer that reaches the *reject* state when the *red_light* turns on, line 4 in Listing 2. The $\{SYS\}1$ prefix indicates the fiacre component where the *red_light* variable is defined. The second requirement, R_2, is represented using an observer automaton that follows the system execution and produces a *success* event whenever the green light is turned on before the *ck* deadline. The observer declaration (line 5) 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 to the **success** state (disabling the timer), and a transition from the **start** state to the **reject** state if the timer expired. These observers are references in the context in which they should be checked and composed with the system during reachability analysis.

4.4 Experimental Results

This section presents some experimental results obtained using our context-aware verification approach [9] on the LGS . All results where obtained using OBP v.1.4.5 on a 64-bit Linux machine that has 64GB of memory.

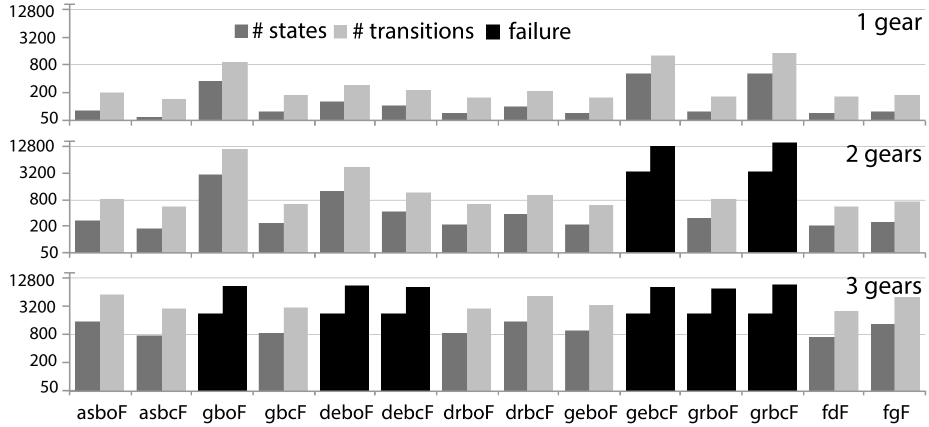


Fig. 5. Reachability analysis results for one/two/three-gear(s) LGS with 1 failure injected interleaved with 3-handle interactions (results in thousands). The black bars indicate the state-space explosion cases, showing the number of states/transitions explored before failure.

While the environment model presented in Sec. 4.2 considers only a single top-level context, our explicit-context modeling approach also enables the analysis of partial system behavior, for instance, by simply running the automatic context split on the *Perturbator* actor we obtain the set of simpler environments that does not take into account the possible model failures. Considering such a context in which the *Pilot* actor sends 3-handle interactions followed by the injection of 2 failures (*drboF*, *fdF*) by the *Perturbator* in a 3 gear system we get a state-space of 1,451,144 states and 4,969,518 transitions in 2,598 sec. However, just by adding one more failure (*geboF*), at the end of the preceding sequence of interactions, the exploration fails (due to lack of memory) after analyzing 1,908,556 states and 6,484,681 transitions.

Even though we could not analyze the whole LGS system using the current version of OBP *Observation Engine*, we have been able to analyze a large number of reachable states of the system. In Fig. 5 we show the results obtained on a

Table 4. Number of sub-context and state-space approximation with respect to the number of gears after two context splitting step

	1 gear	2 gears	3 gears
split 1	391	606	885
split 2	1936	3100	4632
state-space aprox.	1.13×10^8	5.55×10^8	6.72×10^9
Cumulated result for 1 failure with 3 pilot interactions (3-handle)			
# of sub-contexts	14	16	18
states	2 328 635	14 156 119	26 585 225
transitions	5 766 682	53 104 972	98 135 315
time (sec.)	2 387	16 942	46 216

simplification of the model using all 1 failure configurations introduced in Table 3 (the *ldF*, *lgF*, *rdF*, *rgF* are not included in the Fig. since the 1-gear case does not include them, however the results are comparable with *fdF* and *fgF*). Compared to the 1-gear case in the second case the size of the obtained LTS is in average 4.63X(6.44X) bigger in terms of states(transitions), with the *deboF* giving a 10X(15.5X) bigger LTS. It is interesting to note that if in the case of the 2-gear case we reduce the number of *Pilot* interactions to one (1-handle) the size of the resulting LTS drops in average (over the 16-failure cases) by 86.5X(98.4X) states(transitions), with a peak in the case of the *gboF* failure which gives a 146X(174X) smaller system. In the second and third case it should be noted that the 64GB memory space on our machine did not suffice for exploring the context injecting some failures, like *gebcF* and *grbcF* failures. [3] Table 4 shows the number of elementary sub-contexts after one and respectively 2 automatic split levels. The state-space approximation line provides a rough optimistic prediction of the number of reachable states by multiplying the lowest number of states presented in Fig. 5 by the number of sub-context after the second split. The lowest half of Table 4 shows the cumulated results, in terms of LTS size and size, of the exploration of the 1-failure 3-handle contexts shown in Fig. 5.

In Fig. 6 we show a visual representation of the LTS obtained for 3-gear/2 *Pilot* interactions without failures[4]. Two distinct operating modes of the LGS system are shown: at the left we can identify the initialization sequence of the LGS comprising of 7,348 states and 30,605 transitions, while at the right the behavior of the system during a down/up gear sequence is exposed.

Our splitting technique did not suffice for completing the reachability analysis of a 3-gear/3-*Pilot* interactions with failures. However, we argue that despite this setback, the context-aware verification approach introduces a new state-space reduction axis complementary with more holistic approaches such as partial-order reduction [20], and symmetry-reduction [7]. Moreover, the possibility to partially

[3] It should be noted that the instantiation of the model with 1, 2 or 3 gears is can be seen also as a partitioning of the verification on the model-side as opposed to the context-side.

[4] The layout is obtained using Grapviz sfdp layout using a simple linear color scheme where the shorter transitions are red while the longer ones are blue.

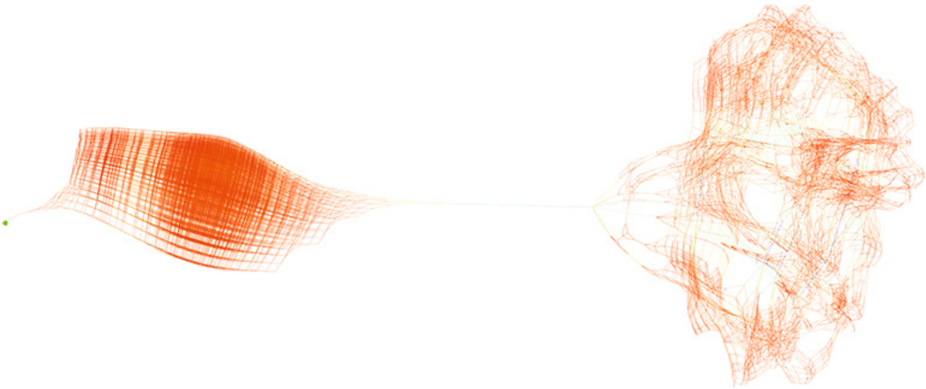


Fig. 6. LGS behaviors during a gear down/up sequence (no failures)

analyze the system gives valuable insights on particular context-dependent behaviors enabling the designer to better focus its verification efforts.

5 Conclusion and Perspectives

In this paper, we apply a novel context-aware verification technique to the Landing Gear System. This approach based on Fiacre and CDL languages and in the OBP *Observation Engine*to OBP proposes to reduce the set of possible behaviors (and thus the state-space) by closing the system-under-study with a well defined environment (context). For LGS we have modeled one top-level context which was automatically decomposed into 885 isolated smaller scenarios, enabling us to iteratively perform reachability analysis on each of them. Even though, some of these scenarios fail due to the state-space explosion problem, we show that our context-aware verification approach pushes the limits of reachability analysis, enabling an automatic divide-and-conquer approach to model-checking. We are currently working on improving our context-aware verification approach by providing a clear methodological framework that formalizes the context coverage with respect to the full system.

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

References

1. Bengtsson, J., Larsen, K.G., Larsson, F., Pettersson, P., Yi, W.: Uppaal — a Tool Suite for Automatic Verification of Real–Time Systems. In: Alur, R., Sontag, E.D., Henzinger, T.A. (eds.) HS 1995. LNCS, vol. 1066, pp. 232–243. Springer, Heidelberg (1996)

2. Boniol, F., Wiels, V.: The Landing Gear System Case Study. In: Boniol, F. (ed.) ABZ 2014 Case Study Track. CCIS, vol. 433, pp. 1–18. Springer, Heidelberg (2014)
3. Boniol, F., Wiels, V., Ledinot, E.: Experiences using model checking to verify real time properties of a landing gear control system. In: Embedded Real-Time Systems (ERTS), Toulouse, France (2006)
4. 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)
5. Clarke, E., Biere, A., Raimi, R., Zhu, Y.: Bounded model checking using satisfiability solving. Formal Methods in System Design 19(1), 7–34 (2001)
6. Clarke, E., Emerson, E., Sistla, A.: Automatic verification of finite-state concurrent systems using temporal logic specifications. ACM Trans. Program. Lang. Syst. 8(2), 244–263 (1986)
7. Clarke, E., Enders, R., Filkorn, T., Jha, S.: Exploiting symmetry in temporal logic model checking. Formal Methods in System Design 9(1-2), 77–104 (1996)
8. 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 2011), Boca Raton, USA (2011)
9. Dhaussy, P., Boniol, F., Roger, J.C., Leroux, L.: Improving model checking with context modelling. Advances in Software Engineering ID 547157, 13 pages (2012)
10. 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: Schürr, A., Selic, B. (eds.) MODELS 2009. LNCS, vol. 5795, pp. 438–452. Springer, Heidelberg (2009)
11. Edelkamp, S., Sanders, P., Šimeček, P.: Semi-external LTL model checking. In: Gupta, A., Malik, S. (eds.) CAV 2008. LNCS, vol. 5123, pp. 530–542. Springer, Heidelberg (2008)
12. 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), SEE, Toulouse (January 2008)
13. Flanagan, C., Qadeer, S.: Thread-modular model checking. In: Ball, T., Rajamani, S.K. (eds.) SPIN 2003. LNCS, vol. 2648, pp. 213–224. Springer, Heidelberg (2003)
14. Godefroid, P.: The Ulg partial-order package for SPIN. In: SPIN Workshop (1995)
15. Holzmann, G.J.: The model checker SPIN. Software Engineering 23(5), 279–295 (1997)
16. Park, S., Kwon, G.: Avoidance of state explosion using dependency analysis in model checking control flow model. In: Gavrilova, M.L., Gervasi, O., Kumar, V., Tan, C.J.K., Taniar, D., Laganá, A., Mun, Y., Choo, H. (eds.) ICCSA 2006. LNCS, vol. 3984, pp. 905–911. Springer, Heidelberg (2006)
17. Peled, D.: Combining Partial-Order Reductions with On-the-fly Model-Checking. In: Dill, D.L. (ed.) CAV 1994. LNCS, vol. 818, pp. 377–390. Springer, Heidelberg (1994)
18. Queille, J.P., Sifakis, J.: Specification and verification of concurrent systems in cesar. In: Dezani-Ciancaglini, M., Montanari, U. (eds.) Programming 1982. LNCS, vol. 137, pp. 337–351. Springer, Heidelberg (1982)
19. Schuppan, V., Biere, A.: Liveness checking as safety checking for infinite state spaces. Electronic Notes in Theoretical Computer Science 149(1), 79–96 (2006)
20. Valmari, A.: Stubborn sets for reduced state space generation. In: Rozenberg, G. (ed.) APN 1990. LNCS, vol. 483, pp. 491–515. Springer, Heidelberg (1991)