HAL
open science

Partially Bounded Context-Aware Verification

Luka Le Roux, Ciprian Teodorov

To cite this version:

Luka Le Roux, Ciprian Teodorov. Partially Bounded Context-Aware Verification. 17th International Conference on Software Engineering and Formal Methods, SEFM 2019, Sep 2019, Oslo, Norway. pp.532-548, 10.1007/978-3-030-30446-1_28 . hal-02434620

HAL Id: hal-02434620

https://hal.science/hal-02434620v1

Submitted on 21 Apr 2020

HAL is a multi-disciplinary open access archive for the deposit and dissemination of scientific research documents, whether they are published or not. The documents may come from teaching and research institutions in France or abroad, or from public or private research centers.

L'archive ouverte pluridisciplinaire **HAL**, est destinée au dépôt et à la diffusion de documents scientifiques de niveau recherche, publiés ou non, émanant des établissements d'enseignement et de recherche français ou étrangers, des laboratoires publics ou privés.

Partially Bounded Context-Aware Verification

Luka Le Roux(✉) and Ciprian Teodorov(✉)

Lab-STICC, MOCS, CNRS UMR 6285, ENSTA Bretagne, Brest, France
{luka.leroux,ciprian.teodorov}@ensta-bretagne.fr

Abstract. Model-checking enables the formal verification of software systems. Powerful and automated, this technique suffers, however, from the state-space explosion problem because of the exponential growth in the number of states with respect to the number of interacting components. To address this problem, the Context-aware Verification (CaV) approach decomposes the verification problem using environment-based guides. This approach improves the scalability but it requires an acyclic specification of the verification guides, which are difficult to specify without losing completeness.

In this paper, we present a new verification strategy that generalises CaV while ensuring the decomposability of the state-space. The approach relies on a language for the specification of the arbitrary guides, which relaxes the acyclicity requirement, and on a partially-bounded verification procedure.

The effectiveness of our approach is showcased through a case-study from the aerospace domain, which shows that the scalability is maintained while easing the conception of the verification guides.

1 Introduction

Since its introduction in the early 1980s, model-checking [11,23] provides an automated formal approach for the verification of complex requirements of hardware and software systems. This technique relies on the exhaustive analysis of all states in the system to check if it correctly implements the specifications, usually expressed using temporal logics. However, because of the internal complexity of the studied systems, model-checking is often challenged with an unmanageable large state-space, a problem known as the state-space explosion problem [8,21]. Numerous techniques [1,7,9,12,28,31] have been proposed to reduce the impact of this problem effectively pushing the inherent limits of model-checking further and further.

Amongst these techniques, the Context-aware Verification (CaV) approach [12,14,16] proposes to separately capture the open system and its environment. From the specifications, the first step of CaV is to formally capture the open system and its *contexts* (environment and property). Each context and the open system are the inputs to a verification task. From there, if one or several tasks

do not scale, CaV offers different automated context-driven techniques for further problem decomposition [13] and for efficient memory management during reachability [26].

This approach was applied to realistic case studies from the medical [5], automotive [25], and aerospace [15,24] domains with very promising results. However, the CaV approach imposes an acyclicity constraint on the verification contexts, which limits expressiveness and renders the approach difficult to use in practice. This limitation impacts the verification engineers who need to manually extract and validate an acyclic model from the environment model. In many cases, the environment behaviours are inherently cyclic and require a verbose and error-prone manual unrolling up to an arbitrarily-chosen depth. Furthermore, when an acyclic model is available, the designer needs to prove its completeness with respects to the complete environment model, problem which is not addressed in the CaV literature.

In this paper, we address these problems through a new verification strategy that generalises CaV. Most notably it enables the specification of cyclic interaction scenarios and uses the closed system as its entry point. The approach is based on an eXtended Guide Description Language (xGDL) and on a partially-bounded verification strategy. The later automatically unrolls these cyclic *verification guides* (previously referred as *context*[1]) to an arbitrary depth. Through this approach the verification engineer is relieved of two tedious tasks: *(a)* extracting the acyclic interaction scenarios from a previously defined environment model, and *(b)* proving the completeness of the extracted scenarios with respect to the full environment model. Moreover, this approach explicitly exposes the *unrolling depth of the verification guides* as a sufficient completeness criteria for the verification. Showing that this bound is sufficient for completeness may be simpler than proving that the length of all paths is sufficient. The core of any model-checking strategy, the reachability analysis, up to the reachability diameter of the system, is necessary for the verification of safety and bounded-liveness properties. In general, our approach aims at the verification of arbitrary properties, however, in the context of this paper we focus on the reachability analysis.

The approach is validated on an aircraft Landing Gear System (LGS), introduced in [6]. Through this case-study we emphasis: *(a)* the usage of xGDL for modelling verification guides, used for closing the system for verification, and for guiding the reachability procedure; *(b)* a state-space decomposition procedure based on the syntactic rewriting of the verification guides, and; *(c)* some reachability results, obtained through the complementarity of our partially-bounded reachability analysis in conjunction with the CaV state-space decomposition strategies.

Section 2 introduces the related work focusing on the CaV approach and its similarities to Bounded Model Checking (BMC). Section 3 describes our main

[1] **Contexts and guides:** CaV uses the open system (no environment) and a *context* as an entry point. The generalisation presented in this paper uses the closed system instead and restrict its environment through a [verification] *guide*.

contribution, the semantics of the guide description language, the partially-bounded verification procedure and discusses the completeness conditions. Section 4 presents LGS system and the associated xGDL model along with the obtained results. Section 5 concludes this study introducing some future research directions.

2 Background and 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 model-checking in the early 1980s [23], several model-checker tools have been developed to help the verification of concurrent systems [2, 18, 31].

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 reacheable states with respect to the number of interacting components. To enable the verification of ever larger systems, numerous research efforts are focused on reducing the impact of the state-space explosion problem. Some of these approaches use efficient data-structures such as BDD [7] for achieving compact state-space representation. Other approaches prune the state-space using techniques such as partial-order reduction [17, 22, 28] and symmetry reduction [9] that exploit fine-grain transition interleaving symmetries and global system symmetries respectively.

Complementary to these, are techniques based on the specification of environments relevant to the studied system [20, 24, 27, 30]. These approaches propose tools that generate environments, based either on assumptions on the system and its interactions with the environment [20, 27], or on the properties that need to be verified [30]. Amongst these, 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 behaviours along with the requirements to be verified. The model is decomposed in two components: the system-under-study and the environment. While the system specification is viewed as a black-box that never changes during the verification, the environment model is decomposed in multiple *acyclic interaction scenarios*, expressed with the Context Definition Language (CDL). The verification contexts are created by associating to each interaction scenario the relevant properties. The verification process iteratively composes these contexts with the system to verify the associated properties. The CaV approach imposes a formal, methodical decomposition and classification of large requirements sets, a first step in overcoming the state-space explosion problem. To guarantee the exhaustiveness of the analysis, the verification should be accompanied by a completeness proof showing that all behaviours unrolled by the guide are sufficient.

CaV relies on CDL formalism to specify the verification guides separately from the system. The core concept of the CDL language is the *context*, which

associates the requirements to be verified to a verification guide (an acyclic component communicating asynchronously with the system). The interaction of the system with the environment is specified through a number of interaction scenarios. The interleaving of these interaction scenarios generates a transition system representing all the bounded behaviours of the environment, which can be fed as input to model-checkers. Moreover, CDL enables the specification of requirements about the system's behaviour as properties that are verified by the OBP Observation Engine. These properties expressed through property-pattern definitions [14] are based on events (e.g. variable x changed), predicates, and synchronous observers.

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 bounded reachability analysis. Hence, in the absence of a full-coverage proof, these approaches cannot guarantee the absence of errors, but only their presence. The usage of explicit acyclic behaviors, and the CaV approach can be considered as the explicit-state equivalent of symbolic BMC. Moreover, as opposed to BMC, the usage of acyclic behaviors offers more flexibility for specifying the "bounds" of the analysis, and the context can be seen as a high-level skeleton which drives the analysis through a complex state-space partition.

The xGDL language, introduced in this study focuses on the specification of the verification guides. This study generalizes the CaV approach by enabling the specification of cyclic verification guides, which releases the need of extracting acyclic models from the environment. Moreover, as opposed to the guide specification in the CDL language, the xGDL specifications are semantically decoupled from the system. During verification, the xGDL specifications are synchronously composed with the system through a labeling function.

By enabling the definition of acyclic verification guides, this study improves the applicability of the CaV approach. Prior to the verification step the verification guides are unrolled to a predefined bound, similarly to BMC. The main difference stems however in the scope of the bound. For BMC the bound is global over the system and its environment, in our approach the bound is partial, applying only to the verification guide.

3 A Language for Context Guided Reachability: xGDL

The approach proposed in this paper supposes a closed transition system as an entry point. By definition, a closed system includes behaviours from both the verification target and its environment. This ensures compatibility with a wide range of verification techniques with the same entry point, independently of the formalism used for property specification.

In addition, our approach requires a labelling function (a total and deterministic relation) over the closed system transitions with the co-domain in $A \cup \{\tau\}$, where A is the set of observable actions involving the environment (referred later as *interactions*) and where τ denotes the lack thereof.

A xGDL specification defines a language over A or a subset of A. The synchronous composition of the closed system and a xGDL specification thus

restricts the sequences of possible interactions to those accepted by the specified language.

Section 3.1 provides the abstract syntax of xGDL, Sect. 3.2 provides its operational semantics through inference rules, Sect. 3.3 explicitly defines the compilation of a xGDL specification to a verification guide (a deterministic finite automaton, DFA), Sect. 3.4 details how a verification guide and the closed transition system to be verified are synchronously composed.

3.1 xGDL Abstract Syntax

A xGDL verification guide defines a language of interactions. Those are drawn from a finite alphabet A. The syntax of xGDL is given by the following BNF-style grammar:

$$\begin{array}{rl} & \bot \mid a \mid C;C \mid C\Box C \mid C\|C \mid \\ C ::= & C? \mid C+ \mid C* \mid C\{i,j\} \mid \\ & \{i,j\}\ of\ [C_1,...,C_n] \end{array}$$

C ranges over the set $\mathcal{E}$ of terms of the xGDL language, a ranges over the alphabet A of observable interactions, and $i, j \in \mathbb{N}$ with $i \leq j$.

According to the previous grammar, an xGDL specification is one of the following: – $\bot$, the empty term; – a, an observable interaction; – $C;C$, a sequential composition of two terms; – $C\Box C$, a non-deterministic choice between two terms; – $C\|C$, a parallel composition, by unrestricted interleaving of two terms; – $C?$, an optional term – $C+$, an unbounded replication of a term, with at least one occurrence; – $C*$, an unbounded replication of a term, with potentially 0 occurrences; – $C\{i,j\}$, a bounded replication of the a term with at least i occurrences and at most j; – $\{i,j\}\ of\ [C_1,...,C_n]$, possible permutations of length at least i to at most j among a set of terms.

3.2 xGDL Operational Semantics

xGDL operational semantics is defined via inference rules. The notation $C \xrightarrow{a} C'$ denotes a tuple $(C, a, C') \in \mathcal{E} \times \{A \cup \tau\} \times \mathcal{E}$, where A is the alphabet of interactions (observable actions initiated by the closed system's environment), τ denotes the lack thereof and $\mathcal{E}$ is the set of all possible terms. If $C \xrightarrow{a} C'$ with $a \neq \tau$, then C can be translated into C' upon *executing* the interaction a. If $C \xrightarrow{\tau} C'$, then C and C' can be said to be semantically equivalent.

$$\frac{a \in A^+}{a \xrightarrow{a} \bot}\ [\text{atom}] \quad \frac{a \in A^+}{a;C \xrightarrow{a} C}\ [\text{seq}_1] \quad \frac{C_1 \xrightarrow{a} C_1' \wedge C_1 \neq a}{C_1;C_2 \xrightarrow{a} C_1';C_2}\ [\text{seq}_2]$$

$$\frac{}{C_1\Box C_2 \xrightarrow{\tau} C_1}\ [\text{alt}_1] \quad \frac{}{C_1\Box C_2 \xrightarrow{\tau} C_2}\ [\text{alt}_2] \quad \frac{C_1 \xrightarrow{a_1} C_1'}{C_1\|C_2 \xrightarrow{a_1} C_1'\|C_2}\ [\text{par}_1]$$

$$\frac{C_2 \xrightarrow{a_2} C_2'}{C_1\|C_2 \xrightarrow{a_2} C_1\|C_2'}\ [\text{par}_2] \quad \frac{}{\bot\|C \xrightarrow{\tau} C}\ [\text{par}_3] \quad \frac{}{C\|\bot \xrightarrow{\tau} C}\ [\text{par}_4]$$

Atom, Sequence, Alternative and Parallelism. If the term is a single **interaction** a, it is *executed* and it results in the empty term $\perp$ ($a \xrightarrow{a} \perp$, rule *atom*).

If the term is a **sequence** of the form $a; C$, the interaction a is *executed* and it results in the term C ($a; C \xrightarrow{a} C$, rule *seq*$_1$). If the term is a sequence of the form $C_1; C_2$ such that C_1 is not a single interaction and such that $\exists (a, C_1') \in \{A \cup \tau\} \times \mathcal{E},\ C_1 \xrightarrow{a} C_1'$, then the interaction a is *executed* and it results in the term $C_1'; C_2$ ($C_1; C_2 \xrightarrow{a} C_1'; C_2$, rule *seq*$_2$).

If the term is a **non-deterministic choice** of the form $C_1 \Box C_2$, it can either result in C_1 ($C_1 \Box C_2 \xrightarrow{\tau} C_1$, rule *alt*$_1$) or C_2 ($C_1 \Box C_2 \xrightarrow{\tau} C_2$, rule *alt*$_2$). In both cases, no interaction is *executed*.

Lastly, if the term is a **parallel composition** of the form $C_1 \| C_2$ with $\exists (a_1, C_1') \in \{A \cup \tau\} \times \mathcal{E},\ C_1 \xrightarrow{a_1} C_1'$ and $\exists (a_2, C_2') \in \{A \cup \tau\} \times \mathcal{E},\ C_2 \xrightarrow{a_2} C_2'$, it can either result in $C_1' \| C_2$ ($C_1 \| C_2 \xrightarrow{a_1} C_1' \| C_2$, rule *par*$_1$) or $C_1 \| C_2'$ ($C_1 \| C_2 \xrightarrow{a_2} C_1 \| C_2'$, rule *par*$_2$) by executing the corresponding interaction. If $C_1 = \perp$ or $C_2 = \perp$, it results in the leftover term (rules *par*$_3$ and *par*$_4$).

$$\frac{}{C? \xrightarrow{\tau} \perp \Box C}\ [\text{opt}] \qquad \frac{}{C* \xrightarrow{\tau} (C; C*)?}\ [\text{star}]$$

$$\frac{}{C+ \xrightarrow{\tau} C; C*}\ [\text{plus}] \qquad \frac{0 < i \leq j}{C\{i,j\} \xrightarrow{\tau} C; C\{i-1, j-1\}}\ [\text{rep}_1]$$

$$\frac{i = 0 \wedge j > 0}{C\{i,j\} \xrightarrow{\tau} (C; C\{0, j-1\})?}\ [\text{rep}_2] \qquad \frac{i = j = 0}{C\{i,j\} \xrightarrow{\tau} \perp}\ [\text{rep}_3]$$

Replications. If the term is an **optional term** of the form $C?$, it is semantically equivalent to $\perp \Box C$, meaning it can either result in $\perp$ or C ($C? \xrightarrow{\tau} \perp \Box C$, rule *opt*).

If the term is an **unbounded replication** of the form $C*$, it is semantically equivalent to $(C; C*)?$ (recursive definition), meaning it results either in $\perp$ or $C; C*$ ($C* \xrightarrow{\tau} (C; C*)?$, rule *star*).

If the term is an **unbounded replication with at least one occurrence** of the form $C+$, it is semantically equivalent to $C; C*$ ($C+ \xrightarrow{\tau} C; C*$, rule plus).

The **bounded replication** $C\{i,j\}$ is defined by the rules *rep*$_1$, *rep*$_2$ and *rep*$_3$. The first applies as long as $i > 0$, decrements both i and j and ensures at least i occurrences of C. The second applies for $i = 0 \wedge j > 0$, decrements j and ensures at most j occurrences of C. The last one applies for $i = j = 0$ and results in $\perp$ (termination).

$$\frac{0 < i \leq j \leq n \wedge \forall k, 1 \leq k \leq n}{\{i,j\}\ of\ [C_1, ..., C_n] \xrightarrow{\tau} C_k; \{i-1, j-1\}\ of\ [C_1, ..., C_{k-1}, C_{k+1}, ..., C_n]}\ [\text{perm}_1]$$

$$\frac{\mathbf{0 = i} < j \leq n \wedge \forall k, 1 \leq k \leq n}{\{i,j\}\ of\ [C_1, ..., C_n] \xrightarrow{\tau} (C_k; \{0, j-1\}\ of\ [C_1, ..., C_{k-1}, C_{k+1}, ..., C_n])\mathbf{?}}\ [\text{perm}_2]$$

$$\frac{0 = i = j}{\{i,j\}\ of\ [C_1, ..., C_n] \xrightarrow{\tau} \perp}\ [\text{perm}_3] \qquad \frac{}{\{i,j\}\ of\ [] \xrightarrow{\tau} \perp}\ [\text{perm}_4]$$

Permutations. The permutation operator, as defined by the above rules, represents the set of possible sequences made of at most one occurrence of each of the terms from the provided set $[C_1, ..., C_n]$ of size i to j (unless $n < i$ or $n < j$, as the size can not exceed n). The notation $[C_1, ..., C_{k-1}, C_{k+1}, ..., C_n]$ (as found in rule $perm_2$) stands for the set $[C_1, ..., C_n]$ minus the term C_k with $i \leq k \leq j$. Rules $perm_3$ and $perm_4$ ensure termination in cases where $j = 0$ and where the set of terms to choose from is empty, respectively.

Prefix Closed Semantics. Defined this way, xGDL syntax and semantics match those of regular expressions extended with parallelism and permutations. However, a xGDL specification defines the language of all possible sequences of interactions. All prefixes of a term accepted by a xGDL specification (including $\perp$) are also members of this language. Thus, unlike regular expressions, xGDL semantics is prefix closed.

3.3 xGDL Compilation

A xGDL specification defines a language over the set of possible interactions A. To ease subsequent manipulations (such as the composition with the closed system as defined Sect. 3.4), a xGDL specification is compiled to a practical verification guide, a deterministic finite automaton (DFA).

Fig. 1. The xGDL compilation flow.

The compilation flow, presented in Fig. 1, starts with a xGDL specification. By applying the semantic rules defined Sect. 3.2 the specification is straightforwardly converted to a non-deterministic finite automaton (NFA). The resulting NFA is then converted to a DFA. For this purpose, transitions carrying no interactions (τ) are considered as ϵ-transitions and are thus removed.

Lastly, this DFA is minimised. The result represents the compiled verification guide. The equivalence between the initial xGDL specification and the compiled verification guide follows directly from well known results in the automaton theory.

3.4 xGDL Guide and Closed System Composition

Given a closed transition system S, a set of interactions A, a labelling function L over $A \cup \{\tau\}$ and a xGDL verification guide G specified over A, the following defines the result their composition.

First, some additional notations are introduced:

- $G \times S$ denotes the resulting transition system;
- G_0, S_0 and $G_0 \times_0 S_0$ denote the initial states set of G, S and $G \times S$;
- (g, s) denotes a composite state;
- $s \xrightarrow{a} s'$ denotes the existence of a transition such that $L(s \rightarrow s') = a$.

Intuitively G and S are seen as transition systems labelled over $A \cup \{\tau\}$ (LTS). $G \times S$ is the result of their synchronous composition with stuttering steps and A as the vocabulary of synchronous behaviours.

The guide LTS G can be obtained through interpretation of a xGDL expression as described by the operational semantics (see Sect. 3.2). However, in the following, the DFA obtained after compilation (see Sect. 3.3) is considered instead. Both are equivalent for this section purpose, but the later being a minimal representation (least possible amount of states) it leads to better exploration results (smaller state space). It also ensures no τ-transitions in G, which eases our definitions.

The system LTS S is obtained by labelling each and every transition t_S from the system under study with $L(t_S) \in A \cup \{\tau\}$. A system transition labelled by $a \in A$ carries the execution of the corresponding interaction. A system transition labelled by τ denotes an *internal* step free of interactions.

The composition $G \times S$ is also a LTS and, as already stated, is obtained by a synchronous composition (over A) with stuttering steps (τ). The following rules define its initial states and transitions:

- Initial states: $(g_0, s_0) \in G_0 \times_0 S_0 \quad \Leftrightarrow \quad g_0 \in G_0 \wedge s_0 \in S_0$;
- Stuttering steps: $(g, s) \xrightarrow{\tau} (g', s') \quad \Leftrightarrow \quad g = g' \quad \wedge s \xrightarrow{\tau} s'$;
- Synchronisations: $a \neq \tau$, $(g, s) \xrightarrow{a} (g', s') \quad \Leftrightarrow \quad g \xrightarrow{a} g' \quad \wedge s \xrightarrow{a} s'$.

Defined as such, G and S mutually constrain one another through their composition. The existence, in the resulting system, of a transition labelled by $a \neq \tau$ from a state (g, s) implies the existence of transitions labelled by a from both g and s.

However, most often in practical cases, all states from S are complete over A. Meaning, for all $a \in A$ and all s a system state, there is a transition from s and labelled by a (possibly modulo some stutters). This is due to A denoting possible interactions with the systems that can be expected at any time. In these cases, S does not constrain G in $G \times S$.

Neutral Guide. Given S, A and L, it is always possible to *build* a neutral guide 1 such that $S = 1 \times S$ (where $=$ denotes a strong bi-simulation).

This can be proven by construction of 1 as the guide with one initial state $\{g_0\}$ and, for all $a \in A$, $g_0 \xrightarrow{a} g_0$. This particular guide follows directly from the xGDL expression $(a_0 \Box a_1 \Box \ ... \ \Box a_{n-1})*$ with $A = \{a_0, a_1, \ ... \ , a_{n-1}\}$.

Subset of Interactions. It is important to note that, unless otherwise specified, the absence of references to an interaction within an xGDL specification *prohibits* that interaction from *happening*.

In cases where the xGDL specification is intended to be defined over a subset $A' \subset A$ of interactions, L (the labelling function) has to be filtered so that it doesn't label transitions by *ignored* interactions (in $A \backslash A'$).

Let L' be this filtered labelling function with $A' \cup \{\tau\}$ as its co-domain, for all t_S (transitions in S):

- $L(t_S) \in A' \cup \{\tau\} \quad \Rightarrow \quad L'(t_S) = L(t_S); \quad (\text{ inside } A' \cup \{\tau\})$
- $L(t_S) \in A \backslash A' \quad \Rightarrow \quad L'(t_S) = \tau \quad (\text{ outside } A' \cup \{\tau\})$

In other words, interaction labels in $A \backslash A'$ are interpreted as τ for the purpose of the composition and thus system transitions labelled by those are allowed to stutter (*to move independently from the guide*).

3.5 Partially Bounded Verification

Using a cyclic verification guide for closing the system is equivalent to the traditional model-checking process, in which the system is closed with an arbitrary environment. The context-aware verification approach showed that model-checking problems can be easily decomposed using acyclic verification guides to significantly improve the scalability of model checking. However, CaV is limited by the acylicity of the verification guides, which are difficult to extract and prove complete. Bounded model checking on the other hand, is more general and can be applied directly to model-checking problems. However in practice it is more often used as test procedure due to the difficulty of proving the completeness of the analysis. Based on the xGDL language, in this section, we propose a partially-bounded verification procedure.

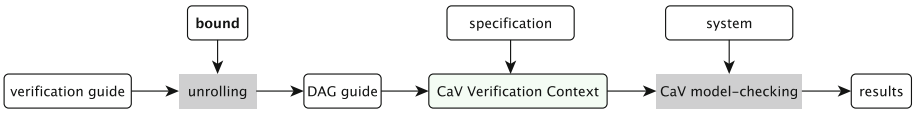

Fig. 2. Partially bounded verification flow

The approach, shown in Fig. 2, is similar to bounded model checking, with the particularity that only the verification guide is bounded. The compiled xGDL guide is unrolled to a predefined *bound*, through this unrolling a directed-acyclic graph (DAG) is obtained satisfying the CaV acyclity requirement. This DAG guide is then associated to a specification to obtain a CaV verification context. The model-checking procedure then analyses this verification context in conjunction with the system (system in the figure). Since the DAG guide is acyclic, both the recursive state-space decomposition and the PastFree[ze] algorithms used by the Context-aware Verification approach, can be applied [26].

It should be noted that, in Fig. 2, the verification guide is unrolled prior to the verification step. This prior unrolling can be seen as the automatic extraction of an acyclic verification guide from an arbitrary environment. This extraction step, required by the CaV approach, was previously implicitly done by the designer during the manual specification of the acyclic verification guide.

Partially Bounded Verification and Completeness. This methodology is generally not complete, in the sense that the unrolling of a system along a bounded interaction scenario potentially implies that some states remain undiscovered (e.g. the states unravelled by a longer scenario). This imposes virtually the same limitation as the bounded model-checking procedures [10]. Namely, that the analysis should be accompanied by a completeness proof showing that the bound b_{guide} chosen for the interaction scenario enables the unrolling of its composition with the system to a depth at least equal to the **Completeness Threshold** $\mathbb{C}$. Moreover, given a cyclic environment and an arbitrary system, $\mathbb{C}$ is an upper bound on b_{guide}. Hence, if the Completeness Threshold of the composition is known it is sufficient, but not necessary, to unroll the cyclic environment model to that depth to achieve completeness.

For the verification of safety properties the completeness threshold is given by the reachability diameter r_d (the minimum number of steps required for reaching all reachable states) [19].

This partially bounded verification procedure effectively generalises the CaV approach to arbitrary systems. Based on this new approach, currently we investigate the possibility to automatically compute the minimal b_{guide} that guarantees that the composition of the interaction scenario with the system reaches the Completeness Threshold, which provides the necessary conditions for the completeness proof.

4 Case-Study: The Landing Gear System

This section showcases xGDL on a realistic case-study from the aerospace domain. In the process, we show that it is well suited for iterative state-space decomposition during model-checking.

The landing gear system (LGS) specification [6] includes three gears, each made of several physical parts. These are specified with (continuous) timed constraints, sensors and possible failures. Retraction and extension sequences can be initiated, interrupted and inverted at any time. This system raises a number of interesting issues during verification, some of which have already been subject to studies via model-checking [4,15,24,26,29].

The focus, here, is not to illustrate how the system can be translated into an executable model. Rather, given that the executable model is already provided, and that the analysis does not scale, this study shows why a language like xGDL is needed and how it can be used within a verification activity requiring several iterations.

Section 4.1 provides an overview of the LGS executable model. Section 4.2 illustrates the definition of the xGDL verification guide and how it can be decomposed, eventually bounded, to further push the limits of the verification.

4.1 LGS Executable Model

The LGS model is composed of the system-under-study along with the capabilities of its environment, both implemented using timed automatons in Fiacre language [3].

System-Under-Study. The LGS manages the extension and retraction of a the landing gears. The physical part includes three landing *boxes* to the front, the left, and the right of the plane. A landing box contains the gear itself as well as a door and hydraulic cylinders. The digital part is responsible of monitoring those physical components through sensors. If an anomaly is detected, this information is forwarded to the cockpit through visual indicators.

A more detailed description of this case study can be found in [6]. The Fiacre implementation of the physical and software parts matches the one proposed and studied via the CAV approach [24,26].

Table 1. Possible failures and labels

Analog Switch		General Electro-Valve	
Opened	Closed	Opened	Closed
f_{1_1}	f_{1_2}	f_{2_1}	f_{2_2}

Door Electro-Valves				Gear Electro-Valves			
Extension		Retraction		Extension		Retraction	
Opened	Closed	Opened	Closed	Opened	Closed	Opened	Closed
f_{3_1}	f_{3_2}	f_{4_1}	f_{4_2}	f_{5_1}	f_{5_2}	f_{6_1}	f_{6_2}

Front	Left	Right	Front	Left	Right
Door			Gear		
f_7	f_8	f_9	f_{10}	f_{11}	f_{12}

Environment Capabilities and System Closure. The pilot can interact with the system through a handle. Switching its position induces *handle* events, which enable the retraction (or extension) sequence.

In addition, a failure may occur at any time. Table 1 lists the possible failures and labels them for future references. Couples $(f_{n_1},\ f_{n_2})$ are exclusive, for example a door may not be blocked in two different positions.

The environment is modelled as one single state automaton in Fiacre. Each of its transitions models a capability, meaning one for the handling of the lever and one per possible failure. This automaton closes the system with its environment capabilities and is later referred as the *system closure automaton.*

Assumptions and Restrictions. The analysis is performed under the following assumptions: *(a)* the software modules are assumed failure-free. *(b)* the sensors, and the interconnect wires are assumed failure-free. *(c)* the failures are assumed permanent, such that if an equipment becomes blocked it remains blocked forever.

Scaling of the Analysis. The resulting state space is much too large[2] for explicit model-checking to scale as is. To address this issue, one can use the fact that at most three failures may happen in one *execution*. If the verification holds for all the valid subsets of three failures, then it holds for the initial problem as well [15]. Taking into account exclusive failures, there is a total of 720 valid subsets and, thus, that many verification tasks.

This can be achieved by various means. Each task can have its own model of the system with different, restricted closure automatons. Parameters can be added to the system and so on. However, these approaches raise new issues regarding the production, soundness, maintainability and further analysis of the various verification tasks. Next section, addresses these issues using the xGDL formalism for the specification of verification guides, which facilitates the decomposition of the state-space while providing the basis for proving its completeness. Moreover, when coupled with the partially-bounded verification procedure, the acyclicity requirement is met, enabling the use of the CAV-specific algorithms.

4.2 xGDL Verification Guides

Specifying the Verification Guide. To apply our approach, an interaction alphabet and a labelling function have to be defined over the executable model introduced in the previous section.

Interaction Alphabet. For this case study, the finite set of interactions considered are inferred from the environment capabilities as described Sect. 4.1. As such:

$$A = \{handle,\ f_{1_1},\ f_{1_2},\ ...,\ f_{6_1},\ f_{6_2},\ f_7,\ ...,\ f_{12}\}$$

Labelling Function. A transition from a system state to another involves zero or one Fiacre transition from the single state automaton modelling the environment capabilities. If present, the labelling function returns the corresponding label. If absent it returns τ, denoting the absence of environment interaction.

xGDL *Guide Expressions.* With the labelling function and its range being now defined, it is possible to write the xGDL expressions. The following introduces some useful examples:

	name		xGDL
- Handles:	G_{pilot}	$=$	$handle\ *$
- One exclusive failure:	$1 \leq n \leq 6,\ F_n$	$=$	$f_{n_1} \Box f_{n_2}$
- One non-exclusive failure:	$7 \leq n \leq 12,\ F_n$	$=$	f_n
- At most three failures:	F_{all}	$=$	$\{0,\ 3\}\ of\ [F_1,\ ...,\ F_{12}]$
- Considered scope:	G_{scope}	$=$	$G_{pilot} \parallel F_{all}$

[2] If the system is restricted to failure-free behaviours, it unfolds 3E+5 states. If restricted to one specific failure, 128Gb of memory is not enough [24,26] (potentially 1E+9 states). For the considered scope (three different failures), those figures hint for a state space several orders of magnitude higher than 1E+10.

G_{pilot} is a sequence of any number of *handle* interactions. The composition of this guide with the system, as defined in Sect. 3.4, induces an analysis restricted to the failure free behaviours (since the failures are not included).

F_n matches one failure injection. For $n \leq 6$, it references a couple of exclusive failures in an alternative so that only one or the other may happen.

F_{all} is a sequence of zero to three failures. The permutation operator is used to ensure uniqueness (a given failure cannot happen twice).

G_{scope} is the parallel composition of G_{pilot} and F_{all}. $G_{scope} \times S$ covers all the possible behaviours minus those outside the specification scope [6] (i.e. at most three unique failures and excludes impossible combinations). In other words, G_{scope} is not strictly neutral to the composition as it is limited to one of each failure and no more than three different ones. However, it precisely and exhaustively captures the system closure required by the specification.

Splitting the Analysis. With the xGDL expressions introduced above, $G_{scope} \times S$ defines the entire state space, target of the verification. As mentioned toward the end of Sect. 4.1, its size is prohibitive for the analysis and needs to be split into smaller, specialised verification tasks.

For this purpose, xGDL can be used to express those through specialised guides. Each of the 720 subsets of three different failures $\{f_i,\ f_j,\ f_k\}$ (with $f_i \neq f_j \neq f_k$) lead to specific xGDL guides:

$$handle * \ \| \ \{0,\ 3\}\ of\ [f_i,\ f_j,\ f_k]$$

Non Intrusive. Using this approach, **the system executable model (S) is an invariant of all the verification tasks**, including the initial one ($G_{scope} \times S$). This approach does not require custom environment closures nor the modification (parameterization) of the system model.

Thus, **one can focus on the languages recognised by the various** xGDL **expressions to provide a soundness proof** of these new verification tasks. To prove that the language of the initial guide is equal to the union of the languages of the guides generated after the splitting process is enough for safety requirements (reachability). For the LGS case study, this is expressed through the Theorem 1 and its proof.

Theorem 1. $language(G_{scope}) = \cup_{id=0}^{719}\ language(G_{id}^3)$
Where $G_{id}^3 = handle * \ \| \ \{0,\ 3\}\ of\ F_{id}^3$ *with* F_0^3 *to* F_{719}^3 *the* 720 *valid subsets of three failures.*

Proof. **By successive rewriting of the equality right hand size:**

0: $\cup_{id=0}^{719}\ language(G_{id}^3)$
1: $language(G_0^3 \ \square \ ... \ \square \ G_{719}^3)$
2: $language(\ (handle * \ \| \ \{0,\ 3\}\ of\ F_0^3) \ \square \ ... \ \square \ (handle * \ \| \ \{0,\ 3\}\ of\ F_7^3 19)\ )$
3: $language(\quad handle * \ \| \ (\{0,\ 3\}\ of\ F_0^3 \ \square \ ... \ \square \ \{0,\ 3\}\ of\ F_{719}^3) \quad)$
4: $language(\quad handle * \ \| \ \{0,\ 3\}\ of\ F_{all} \quad)$
5: $language(G_{scope})$

Step 0 to 1: the union of xGDL expressions languages is equal to the language of an alternative over those expressions. Step 1 to 2: unfolding of all the G_i^3. Step 2 to 3: the parallel operator is distributive over the alternative (i.e. $(A\|B)\Box(A\|C) = A\|(B\Box C)$) Step 3 to 4: the alternative over the length three permutations of F_{id}^3 subsets is equal to F_{all} (both strictly recognise all the valid, length three permutations). Step 4 to 5: per definition of G_{scope}. *QED.*

Further Refinements and Bounding the Verification. Model-checking of any of the 720 guides with three specific failures still did not scale. xGDL offers the possibility to further refine the verification guides and to partially bound the verification tasks in the guide specifications.

Table 2. Unrolling bounds required for completeness

Failure	f_{1_1}	f_{1_2}	f_{2_1}	f_{2_2}	f_{3_1}	f_{3_2}	f_{4_1}	f_{4_1}	f_{5_1}	f_{5_2}	f_{6_1}	f_{6_2}	$f_7,\ f_8,\ f_9$	$f_{10},\ f_{11},\ f_{12}$
Bound	16	16	18	17	20	20	18	20	20	X	18	X	20	20

Similarly to the guide with up to three failures, a guide including exactly one failure ($handle* \ \| \ \{1,\ 1\}\ of\ [F_1,\ ...,\ F_{12}]$) can be split into 18 guides:

$$G_i^1 = handle* \ \| \ f_i$$

Bounding those 18 verification tasks (as shown in Sect. 3.5) arbitrarily to 30 interactions allows the analysis to successfully terminate for 16 of these. To prove completeness, an option is to perform an analysis of the induced clusters of states, as discussed in [26]. In this case, a cyclic behaviour is detected after 16 to 20 interactions depending on the considered failure. Table 2 shows, for each failure, the bound required for completeness inferred from this post-mortem analysis.

For failures f_{5_2} and f_{6_2} (extension and retraction gear electro-valves blocked in closed position), further refinement is still needed. Since G_i^1 has exactly one interaction on the right hand side of the parallel operator, it is equivalent to the sequence:

$$G_i^1 = handle* \ ; \ f_i \ ; \ handle*$$

Additionally, for $handle*$, bounding the analysis and inferring the bound required for completeness shows 7 handles are enough to consider before the eventual failure. This can be captured as:

$$G_i^1 \Leftrightarrow handle\{0,\ 7\} \ ; \ f_i \ ; \ handle*$$

This last form can then be decomposed again in 16 different guides of the form $handle\{n, n\} \ ; \ f_i \ ; \ handle*$ with $0 \leq n \leq 7$ and $f_i \in \{f_{5_2},\ f_{6_2}\}$. For both failures, the analysis bounded to 30 interactions scales for $0 \leq n \leq 4$ but would still require further refinement for $n > 4$.

The approach proposed in this study allows the use complementary analysis techniques on the same executable model and its properties. From this perspective, typically several directions are possible, such as: 1. further abstracting the model for symbolic model-checking; 2. exploit symmetry reduction or partial-order reduction.

Producing the various verification tasks is done without altering the formal specification of the initial challenge. Moreover, xGDL enables to dispatch the verification tasks to different, complementary tools.

5 Conclusion and Perspectives

This paper presented a guide description language along with a partially-bounded context-aware verification procedure. Through the xGDL specifications the acyclicity requirement imposed by the CaV methodology is lifted, which bridges the gap between the environment model and the verification guides. These cyclic verification guides are unrolled to a predefined depth before their composition with the system, which enables the use of the CaV state-space decomposition algorithms. The approach was illustrated on a landing gear system case study. The system/environment interactions where formally captured using one xGDL guide. Relying on this guide, the verification problem was decomposed in 720 sub-problems. This decomposition is accompanied by a coverage proof realised by rewriting of the guide structure. Most of the one-failure cases (16 out of 18) where discharged using the partially-bounded verification procedure, which used in conjunction with the PastFree algorithm of CaV provided the completeness proof, by bi-simulation on the clusters induced by the guide. The two failing guides, were further rewritten and decomposed (structurally), and the new form was partially-bounded (syntactically) using the completeness threshold of the failure free analysis. Currently, we are investigating an online verification procedure, which unrolls the guide during the verification while at the same time enabling the recursive state-space decomposition.

References

1. Barnat, J., Brim, L., Simecek, P.: Cluster-based I/O-efficient LTL model checking. In: Proceedings of the 2009 IEEE/ACM International Conference on Automated Software Engineering, pp. 635–639. ASE 2009. IEEE Computer Society, Washington (2009). https://doi.org/10.1109/ASE.2009.32
2. Bengtsson, J., Larsen, K., Larsson, F., Pettersson, P., Yi, W.: UPPAAL — a tool suite for automatic verification of real-time systems. In: Alur, R., Henzinger, T.A., Sontag, E.D. (eds.) HS 1995. LNCS, vol. 1066, pp. 232–243. Springer, Heidelberg (1996). https://doi.org/10.1007/BFb0020949
3. Berthomieu, B., et al.: Fiacre: an intermediate language for model verification in the topcased environment. In: European Congress on Embedded Real-Time Software (ERTS). SEE, Toulouse, France (Jan 2008). https://hal.inria.fr/inria-00262442

4. Berthomieu, B., Dal Zilio, S., Fronc, Ł.: Model-checking real-time properties of an aircraft landing gear system using fiacre. In: Boniol, F., Wiels, V., Ait Ameur, Y., Schewe, K.-D. (eds.) ABZ 2014. CCIS, vol. 433, pp. 110–125. Springer, Cham (2014). https://doi.org/10.1007/978-3-319-07512-9_8
5. Boniol, F., Dhaussy, P., Le Roux, L., Roger, J.C.: Model-based analysis. In: Embedded systems, Analysis and Modeling with SysML, UML and AADL, pp. 157–184. Wiley (May 2013). https://hal.archives-ouvertes.fr/hal-00843139
6. Boniol, F., Wiels, V.: The landing gear system case study. In: Boniol, F., Wiels, V., Ait Ameur, Y., Schewe, K.-D. (eds.) ABZ 2014. CCIS, vol. 433, pp. 1–18. Springer, Cham (2014). https://doi.org/10.1007/978-3-319-07512-9_1
7. Burch, J., Clarke, E., McMillan, K., Dill, D., Hwang, L.: Symbolic model checking: 10^{20} states and beyond. Inf. Comput. **98**(2), 142–170 (1992). https://doi.org/10.1016/0890-5401(92)90017-A
8. 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). https://doi.org/10.1145/5397.5399
9. Clarke, E.M., Enders, R., Filkorn, T., Jha, S.: Exploiting symmetry in temporal logic model checking. Formal Methods Syst. Des. **9**(1), 77–104 (1996). https://doi.org/10.1007/BF00625969
10. Clarke, E., Biere, A., Raimi, R., Zhu, Y.: Bounded model checking using satisfiability solving. Formal Methods Syst. Des. **19**(1), 7–34 (2001). https://doi.org/10.1023/A:1011276507260
11. Clarke, E.M., Emerson, E.A.: Design and synthesis of synchronization skeletons using branching time temporal logic. In: Kozen, D. (ed.) Logic of Programs 1981. LNCS, vol. 131, pp. 52–71. Springer, Heidelberg (1982). https://doi.org/10.1007/BFb0025774
12. Dhaussy, P., Boniol, F., Landel, E.: Using context descriptions and property definition patterns for software formal verification. In: Proceedings of the 2008 IEEE International Conference on Software Testing Verification and Validation Workshop, pp. 89–96. ICSTW 2008. IEEE Computer Society, Washington (2008). https://doi.org/10.1109/ICSTW.2008.52
13. Dhaussy, P., Boniol, F., Roger, J.C., Le Roux, L.: Improving model checking with context modelling. In: Advances in Software Engineering 2012, ID 547157, 13 p (October 2012). https://doi.org/10.1155/2012/547157
14. 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). https://doi.org/10.1007/978-3-642-04425-0_34
15. Dhaussy, P., Teodorov, C.: Context-aware verification of a landing gear system. In: Boniol, F., Wiels, V., Ait Ameur, Y., Schewe, K.-D. (eds.) ABZ 2014. CCIS, vol. 433, pp. 52–65. Springer, Cham (2014). https://doi.org/10.1007/978-3-319-07512-9_4
16. Dumas, X., Dhaussy, P., Boniol, F., Bonnafous, E.: Application of partial-order methods for the verification of closed-loop SDL systems. In: Proceedings of the 2011 ACM Symposium on Applied Computing, pp. 1666–1673. SAC 2011. ACM, New York (2011). https://doi.org/10.1145/1982185.1982533
17. Godefroid, P.: The Ulg partial-order package for SPIN. In: SPIN Workshop. Montréal, Quebec (1995)
18. Holzmann, G.J.: The model checker SPIN. IEEE Trans. Softw. Eng. **23**(5), 279–295 (1997). https://doi.org/10.1109/32.588521

19. Kroening, D., Strichman, O.: Efficient computation of recurrence diameters. In: Zuck, L.D., Attie, P.C., Cortesi, A., Mukhopadhyay, S. (eds.) VMCAI 2003. LNCS, vol. 2575, pp. 298–309. Springer, Heidelberg (2003). https://doi.org/10.1007/3-540-36384-X_24
20. Parizek, P., Plasil, F.: Specification and generation of environment for model checking of software components. Electron. Notes Theor. Comput. Sci. **176**(2), 143–154 (2007). https://doi.org/10.1016/j.entcs.2006.02.036
21. Park, S., Kwon, G.: Avoidance of state explosion using dependency analysis in model checking control flow model. In: Gavrilova, M.L., et al. (eds.) ICCSA 2006. LNCS, vol. 3984, pp. 905–911. Springer, Heidelberg (2006). https://doi.org/10.1007/11751649_99
22. Peled, D.: Combining partial order reductions with on-the-fly model-checking. Formal Methods Syst. Des. **8**(1), 39–64 (1996). https://doi.org/10.1007/BF00121262
23. 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). https://doi.org/10.1007/3-540-11494-7_22
24. Teodorov, C., Dhaussy, P., Le Roux, L.: Environment-driven reachability for timed systems. Int. J. Softw. Tools Technol. Transfer **19**(2), 229–245 (2017). https://doi.org/10.1007/s10009-015-0401-2
25. 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.) MEDI 2014. LNCS, vol. 8748, pp. 53–64. Springer, Cham (2014). https://doi.org/10.1007/978-3-319-11587-0_7
26. Teodorov, C., Le Roux, L., Drey, Z., Dhaussy, P.: Past-free[ze] reachability analysis: reaching further with DAG-directed exhaustive state-space analysis. Softw. Test. Verif. Reliab. **26**(7), 516–542 (2016). https://doi.org/10.1002/stvr.1611
27. Tkachuk, O., Dwyer, M.B.: Environment generation for validating event-driven software using model checking. IET Softw. **4**(3), 194–209 (2010). https://doi.org/10.1049/iet-sen.2009.0017
28. Valmari, A.: Stubborn sets for reduced state space generation. In: Rozenberg, G. (ed.) ICATPN 1989. LNCS, vol. 483, pp. 491–515. Springer, Heidelberg (1991). https://doi.org/10.1007/3-540-53863-1_36
29. Wiels, V., Ledinot, E., Belin, E., Dassault, M.: Experiences in using model checking to verify real time properties of a landing gear control system. In: Embedded Real-Time Systems (ERTS). Toulouse, France (Jan 2006)
30. Yatake, K., Aoki, T.: Automatic generation of model checking scripts based on environment modeling. In: Model Checking Software - 17th International SPIN Workshop, Enschede, The Netherlands, September 27–29, 2010. Proceedings, pp. 58–75 (2010). https://doi.org/10.1007/978-3-642-16164-3_5
31. Yu, Y., Manolios, P., Lamport, L.: Model checking TLA$^+$ specifications. In: Pierre, L., Kropf, T. (eds.) CHARME 1999. LNCS, vol. 1703, pp. 54–66. Springer, Heidelberg (1999). https://doi.org/10.1007/3-540-48153-2_6