Hindawi Publishing Corporation
Advances in Software Engineering
Volume 2012, Article ID 547157, 13 pages
doi:10.1155/2012/547157

Research Article

Improving Model Checking with Context Modelling

Philippe Dhaussy,[1] Frédéric Boniol,[2] Jean-Charles Roger,[1] and Luka Leroux[1]

[1] *Lab-STICC, UMR CNRS 6285, ENSTA Bretagne, 2 rue François Verny, 29806 Brest, France*
[2] *ONERA, 2 avenue Edouard Belin, 31000 Toulouse, France*

Correspondence should be addressed to Philippe Dhaussy, philippe.dhaussy@ensta-bretagne.fr

Received 6 February 2012; Revised 18 May 2012; Accepted 22 May 2012

Academic Editor: Gerardo Canfora

This paper deals with the problem of the usage of formal techniques, based on model checking, where models are large and formal verification techniques face the combinatorial explosion issue. The goal of the approach is to express and verify requirements relative to certain context situations. The idea is to unroll the context into several scenarios and successively compose each scenario with the system and verify the resulting composition. We propose to specify the context in which the behavior occurs using a language called CDL (*Context Description Language*), based on activity and message sequence diagrams. The properties to be verified are specified with textual patterns and attached to specific regions in the context. The central idea is to automatically split each identified context into a set of smaller subcontexts and to compose them with the model to be validated. For that, we have implemented a recursive splitting algorithm in our toolset OBP (Observer-based Prover). This paper shows how this combinatorial explosion could be reduced by specifying the environment of the system to be validated.

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. Despite technical improvements, the increasing size of the systems makes the introduction of a wide range of potential errors easier. 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 difficulty, 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. Several model checkers as [1–5] have been developed to help the verification of concurrent asynchronous systems. It is well known that an important issue that limits the application of model checking techniques in industrial software projects is the combinatorial explosion problem [6–8]. Because of the internal complexity of developed software, model checking of requirements over the system behavioral models could lead to an unmanageable state space.

The approach described in this paper presents an exploratory work to provide solutions to the problems mentioned above. The proposed approach consists to reduce the set of possible behaviors (and then indirectly the state space) by closing the system under verification with a well-defined environment. For this, we propose to specify the behavior of the entities that compose the system environment. These entities interact with the system. These behaviors are described by use cases (scenarios) called here *contexts*. They describe how the environment interacts with the system. Indeed, in the context of embedded reactive systems, the environment of each system is finite and well known. We claim that it is more efficient to ask the engineers to explicitly and formally express this context, than to search to reduce the state space of the system to explore facing an unspecified environment. In other words, the objective is to circumvent the problem of the combinatorial explosion by restricting the system behavior with a specific surrounding environment describing the different configurations in which one wants to verify the system. Moreover, properties are often related to specific use cases (such as initialization, reconfiguration, and degraded modes) so that it is not necessary for a given property to take into account all possible behaviors of the environment, but only the subpart concerned by the verification. The context description thus allows a first limitation of the explored space search, and hence a first reduction of the combinatorial explosion.The second idea exploited is that, if the context is finite (i.e., there is a noninfinite loop in the context) and in case of safety (invariant) properties, then the two following verification processes are equivalent: (a) compose the context and the system, and then verify the resulting global system; (b) unroll the context into N scenarios (i.e., a sequence of events), and successively compose each scenario with the system and verify the resulting composition. In other words, the global verification problem can be transformed into N smaller verification subproblems.

Our approach is based on these two ideas. This paper presents a DSL (domain-specific language). called CDL (*Context Description Language*) for formally describing the environment of the system to be verified. This language serves to support our approach to reduce the state space. We illustrate our reduction technique with our OBP (*Observer-based Prover*) (OBP is available on http://www.obpcdl.org/.) tool connected to two tools: the first is an academic model checker TINA-SELT (http://projects.laas.fr/tina/) [3] and the second is an explorer called OBP Explorer, integrated in OBP. We illustrate our approach with a partial case study provided by a industrial partner in the aeronautics domain.

This paper is organized as follows: Section 2 presents the related techniques to improve model checking by state reduction. Section 3 presents the principles of our approach for context aware formal verification. Section 4 describes the CDL language for contexts specification and property specification. Our toolset used for the experiments is presented Section 5. In Section 6, we give results on the industrial case study. In Section 7, we discuss our approach and we conclude.

2. Related Works

Model checking is a technique that relies on building a finite model of a system of interest, and checking that a desired property, specified as a temporal logic formula, holds in that model. Since the introduction of this technology in the early 1980s, several model checkers have been developed to help the verification of concurrent asynchronous systems. For example, the SPIN model checker [1] based on the formal language PROMELA allows the verification of LTL properties encoded in “never claim” formalism and further converted into Buchi automata. Since its introduction, model checking has advanced significantly. For instance the state compression method or partial-order reduction contributed to the further alleviation of combinatorial explosion [9]. In [10], the partial-order algorithm based on a depth-first search (DFS) has been adapted to the breadth first search (BFS) algorithm in the SPIN model checker to exploit interesting properties inherent to the BFS. Partial-order methods [9, 11, 12] aim at eliminating equivalent sequences of transitions in the global state space without modifying the falsity of the property under verification. These methods, exploiting the symmetries of the systems, seemed to be interesting and were integrated into many verification tools (for instance SPIN).

In the same way, the development of more efficient data structure, such as binary decision diagrams (BDD) [13], allows for automatic and exhaustive analysis of finite state models with several thousands of components or state variables.

Another approach deals with compositional verification, for example, assume/guarantee reasoning or design-by-contract techniques. A lot of work exist in applying these techniques to model checking including, for example, [14–17]. These works deal with model checking/analyzing individual components (rather than whole systems) by specifying, considering, or even automatically determining the interactions that a component has or could have with its environment so that the analysis can be restricted to these interactions. Design-by-contract proposes to verify a system by verifying all its components one by one. Using a specific composition operator preserving properties, it allows assuming that the system is verified.

Many other techniques have been proposed for combating state explosion. On-the-fly verification constructs the state space in a demand-driven way, thus allowing the detection of errors without a priori building the entire state space. Distributed verification [18] uses the computing resources of several machines connected by a network, thus allowing to scale up the capabilities of verification tools. In the same objective, methods exploiting heuristic search [19] have been proposed for improving constraint satisfaction problem and more generally for optimizing the exploration for the behaviour of a model to verify.

Combined together, the successful application of these methods to several case studies (see for instance [20] for noncritical application, or [21, 22] for aerospace examples) demonstrates their maturity in the case of synchronous embedded systems. However, if these techniques are useful to find modelling errors, they still suffer from combinatorial explosion in the case of large and complex asynchronous systems (see [23] for an experiment of SPIN on a real asynchronous function showing that the verification does not complete despite all the optimizations mentioned above).

Our approach presented in this paper explores another way for reducing the combinatorial explosion. Conversely to “traditional” techniques in which contexts are often included in the system model, we choose to explicit contexts separately from the model. It is about using the knowledge of the environment of a whole system (or model) to conduct a verification to the end. We propose to formally specify the context behavior in a way that allows a fully automatic divide-and-conquer algorithm.

Another difficulty is about requirement specification. Embedded software systems integrate more and more advanced features, such as complex data structures, recursion, and multithreading. Despite the increased level of automation, users of finite-state verification tools are still constrained to specify the system requirements in their specification language which is often informal. While temporal logic-based languages (example LTL or CTL [6]) allow a great expressivity for the properties, these languages are not adapted to practically describe most of the requirements expressed in industrial analysis documents. Modal and temporal logics are rather rudimentary formalisms for expressing requirements, that is, they are designed having in mind the straight for wardness of its processing by a tool such as a model checker rather than the user-friendliness. Their efficient use in practice is hampered by the difficulty to write logic formula correctly without extensive expertise in the idioms of the specification languages.

In literature, many approaches have been proposed to enable software and hardware engineers to use temporal logic with ease and rigor. For instance [24, 25] proposed a graphical interval logic (RTGIL) allowing visual an intuitive reasoning on real-time systems. From a textual point of view, [26–28] proposed to formulate requirements using textual patterns, that is, textual templates that capture common logical and temporal properties and that can be instantiated in a specific context. They represent commonly occurring types of real-time properties found in several requirement documents for embedded systems. These two approaches have been recently combined by De Francesco et al. in [29]. The authors propose a user-friendly interface with the aim of simplifying the writing of concurrent system properties. This interface supplies a set of patterns from the natural language which are then automatically translated into the mu-calculus temporal logic.

In this paper, we choose to follow this approach. In order to be as simple as possible, we only consider safety properties expressed by using an extension of textual Dwyer's patterns and translated into observer automata and invariants. The work could be extended to other types or properties as proposed in [29]. Such an extension is out of the scope of this article.

3. Context Aware Verification

To illustrate the explosion problem, let us consider the example in Figure 1. We are trying to verify some requirements by model checking using the TINA-SELT model checker [3] and OBP Explorer. We present the results for a part of the *S_CP* model, which consists in reducing the set of. Then, we introduce our approach based on context specifications.

3.1. An Illustration. We present one part of an industrial case study: the software part of an antiaircraft system (*S_CP*). This controller controls the internal modes, the system physical devices (sensors, actuators) and their actions in response to incoming signals from the environment. The *S_CP* system

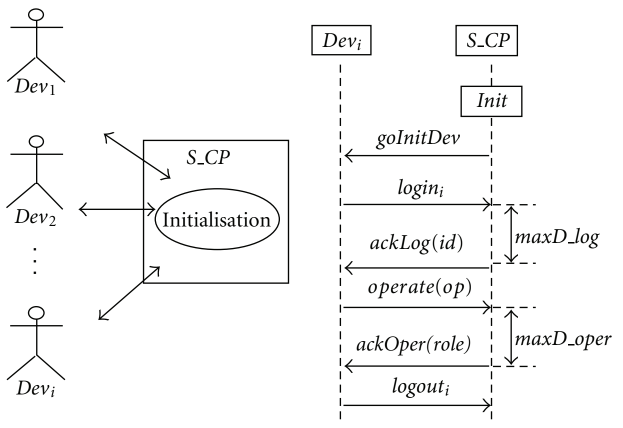


FIGURE 1: *S_CP* system: partial description during the initialization phase.

interacts with devices (*Dev*) that are considered to be *actors* included in the *S_CP* environment called here *context*.

The sequence diagrams of Figure 2 illustrate interactions between context actors and the *S_CP* system during an initialization phase. This context describes the environment we want to consider for the validation of the *S_CP* controller. This context is composed of several actors *Dev* running in parallel or in sequence. All these actors interleave their behavior. After the initializing phase, all actors Dev_i ($i \in [1, \ldots, n]$) wait for orders *goInitDev* from the system. Then, actors Dev_i send $login_i$ and receive either $ackLog(id)$ (Figures 2(a) and 2(c)) or $nackLog(err)$ (Figure 2(b)) as responses from the system. The logged devices can send *operate(op)* (Figures 2(a) and 2(c)) and receive either $ackOper(role)$ (Figure 2(a)) or $nackOper(err)$ (Figure 2(c)). The messages *goInitDev* can be received in parallel in any order. However, the delay between messages $login_i$ and $ackLog(id)$ (Figure 1) is constrained by *maxD_log*. The delay between messages $operate(op)$ and $ackOper(role)$ (Figure 1) is constrained by *maxD_oper*. And finally all Dev_i send $logout_i$ to end the interaction with the *S_CP* controller.

As example, let's see two requirements on the *S_CP* system. These requirements were found in a document of our partner and are shown in Listings 1 and 2.

The first requirement *R*1 is expressed by Listing 1.

We choose to specify this requirement with SELT language for the device *Dev_*1. It is expressed by the following formula:

```
Inv1: []((SM_1_voperateAccepted1) =>
(SM_1_vdevLogged1));
```

*SM_*1 is a process of *S_CP* and *operateAccepted*1 and *devLogged*1 are variables of this process. To verify this requirement, we used the TINA-SELT model checker (Figure 3).

Let's see in Listing 2, the second requirement *R*2.

We choose to specify this requirement with an observer automaton (Figure 4). An observer is an automaton which *observes* the set of events exchanged by the system $\mathcal{S}$ and its context *C* (and thus events occurring in the executions (*runs*) and which produces an event *reject* whenever the

Table 1: Table highlighting the verification complexity for the case study with TINA-SELT.

N (Number of devices)	Exploration and model checking time (sec)	Number of LTS configurations	Number of LTS transitions
1	1	22,977	103,354
2	3	172,095	759,094
3	10	718,623	3,127,468
4	27	2,174,997	9,371,560
5	Explosion	—	—

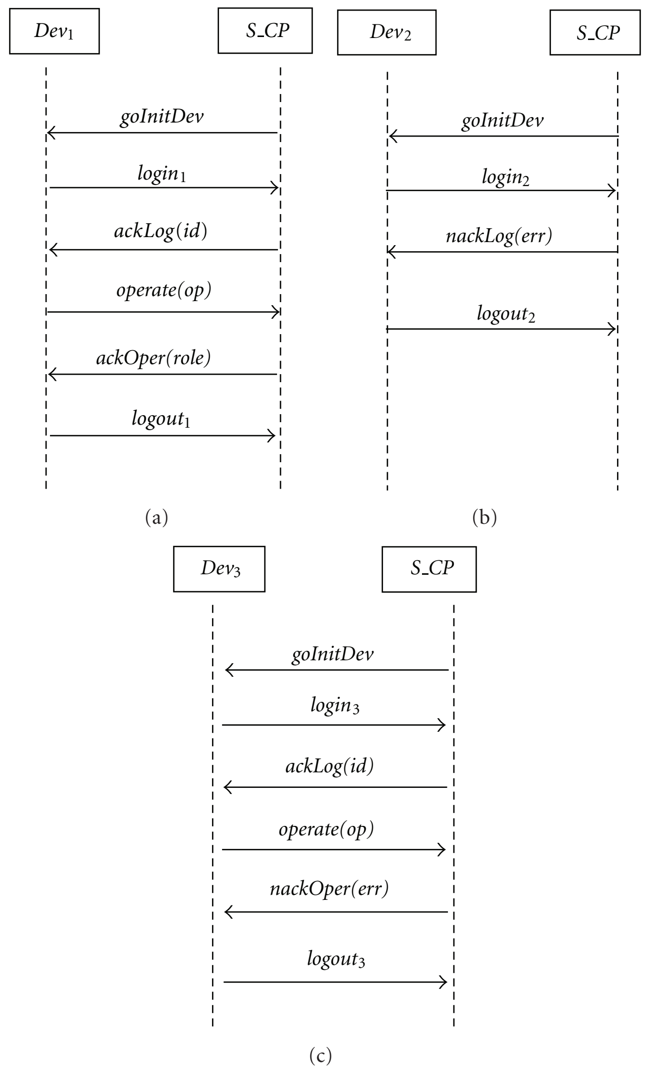


Figure 2: An example of *S_CP* context scenario with 3 devices.

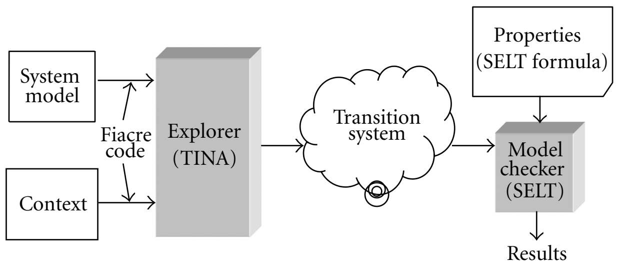


Figure 3: Verification with TINA-SELT.

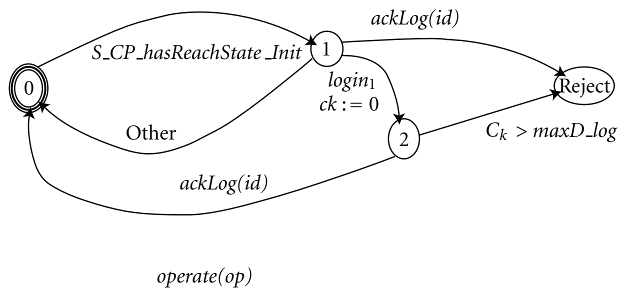


Figure 4: Observer automaton for requirement $R2$.

property becomes false. With observers, the properties we can handle are of safety and bounded liveness type. The accessibility analysis consists of checking if there is a reject state reached by a property observer. In our example, this reject node is reached after detecting the event sequence of *S_CP_hasReachState_Init* and $login_1$, in that order, if the sequence of one or more of *ackLog* is not produced before *maxD_log* time units. Conversely, the reject node is not reached either if *S_CP_hasReachState_Init* or $login_1$ are never received, or if *ackLog* event above is correctly produced with the right delay. Consequently, such a property can be verified by using reachability analysis implemented in our OBP Explorer.

This observer is checked with OBP Explorer (Figure 5).

In both cases, the system model (Here by system model, we refer to the model to be validated) is translated into Fiacre format [30] to explore all the *S_CP* model behaviors by simulation, *S_CP* interacting with its environment (devices). Model exploration generates a labeled transition system (LTS) which represents all the behaviors of the controller in its environment.

3.2. Model Checking Results. Table 1 shows (tests were executed on Linux 32 bits—3 Go RAM computer, with TINA vers.2.9.8 and Frac parser vers.1.6.2.) the TINA-SELT exploration time and the amount of configurations and transitions in the LTS for different complexities (N indicates the number of considered actors). Over four devices, we see a state explosion because of the limited memory of our computer.

Table 2 shows (tests were executed on Linux 32 bits—3 Go RAM computer, with OBP Explorer vers.1.0.) the OBP Explorer exploration analyze time and the amount of configurations and transitions in the LTS. Over three devices, *R1: A device (Dev) can be authorized to execute a command "operate" if it has previously connected to the system.*

Listing 1: Permission requirement for command "operate".

R2: During initialization procedure, S_CP shall associate an identifier to each device (Dev), after login request and before maxD_log time units.

Listing 2: Initialization requirement for the *S_CP* system.

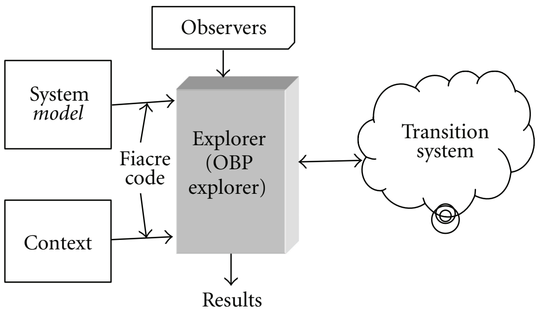


Figure 5: Verification with OBP Explorer.

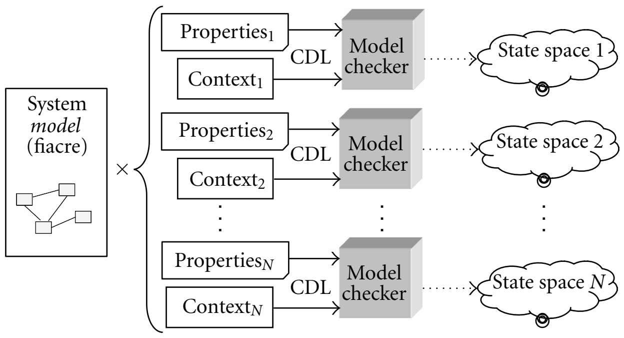


Figure 6: Context-aware model checking.

we see also a state explosion because of the limited memory of our computer.

Note that the size of the LTS explored by OBP Explorer for verifying *R*2 is greater than the size of the related LTS explored by TINA-SELT for verifying *R*1. This is due to the way chosen for modeling these two requirements. *R*1 is formalized as a SELT formula, and *R*2 is modeled as an observer automaton. In the second experiment (*R*2 with OBP Explorer), the explorer begins by building the synchronized product between the model of the system, each context and the observer automaton. If this automaton contains several locations and several clocks, taking into account the observer as an input of the synchronized product could significantly increase the number of states and transitions explored.

3.3. Combinatorial Explosion Reduction. When checking the properties, a model checker explores all the model behaviors and checks whether the properties are true or not. Most of the time, as shown by previous results, the number of reachable configurations is too large to be contained in memory (Figures 3 and 5). We propose to restrict model behavior by composing it with an environment that interacts with the model. The environment enables a subset of the behavior of the model. This technique can reduce the complexity of the exploration by limiting the scope of the verification to precise system behaviors related to some specific environmental conditions.

This reduction is computed in two stages: contexts are first identified by the user ($context_i$, $i \in [1, \ldots, N]$ in Figure 6). They correspond to patterns of use of the component being modeled. The aim is to circumvent the combinatorial explosion by restricting the behavior system with an environment describing different configurations in which one wishes check the requirements. Then each context $context_i$ is automatically partitioned into a set of subcontexts. Here, we precisely define these two aspects implemented in our approach.

3.3.1. Context Identification. The context identification focuses on a subset of behavior and a subset of properties. In the context of reactive embedded systems, the environment of each component of a system is often well known. It is therefore more effective to identify this environment than trying reduce the configuration space of the model system to explore. The proof relevance is based on a strong hypothesis: *it is possible to specify the sets of bounded behaviors in a complete way.* This hypothesis not formally justified in our work. But, in this approach, the essential idea is *the designer can correctly develop a software only if he knows the constraints of its use.* So, we suppose that the designer is able to identify all possible interactions between the system and its environment.

It's particularly true in the field of embedded systems, with the fact that the designer of a software component needs to know precisely and completely the perimeter (constraints, conditions) of its system for properly developing it.

Table 2: Table highlighting the verification complexity for the case study with OBP Explorer.

N (Number of devices)	Exploration and analyze time (sec)	Number of LTS configurations	Number of LTS transitions
1	1	43,828	321,002
2	4	350,256	2,475,392
3	19	1,466,934	6,430,265
4	Explosion	—	—

We also consider second hypothesis. It expresses that the contexts we describe are finite. There are no infinite loops in the interactions between the system and its environment. It is particularly true for instance with command systems or communication protocols.

It would be necessary to study formally the validity of these working hypothesis based on the targeted applications. In this paper, we do not address this aspect that gives rise to a methodological work to be undertaken.

Moreover, properties are often related to specific use cases (such as initialization, reconfiguration, and degraded modes). Therefore, it is not necessary for a given property to take into account all possible behaviors of the environment, but only the subpart concerned by the verification. The context description thus allows a first limitation of the explored space search, and hence a first reduction in the combinatorial explosion.

3.3.2. Context Automatic Splitting. The second idea is to automatically split each identified context into a set of smaller subcontexts (Figure 7). The principle of splitting is as following: each context is represented by an acyclic graph as mentioned earlier. This graph is composed with the model for exploration. In case of explosion, this context is automatically split into several parts taking into account a parameter for the depth in the graph for splitting until the exploration succeeds.

To reach that goal, we implemented a recursive splitting algorithm in our OBP tool. Figure 7 illustrates the function $explore\_mc()$ for exploration of a *model*, with a $context_i$ and model checking of a set of properties pty.

We illustrate one execution of this algorithm in Figures 8 and 9. One context $context_i$, represented by an acyclic graph, is composed with the model S for exploration. In case of explosion, $context_i$ is automatically split into several parts (taking into account a parameter d which specifies the depth in the graph for splitting) until the exploration succeeds. For example in Figure 8, the graph of $context_i$ is split in four graphs $context_{i1}$; $context_{i2}$, $context_{i3}$, and $context_{i4}$. After splitting of $context_i$, the subcontexts are composed with the model for exploration. If exploration fails, one subcontext is split, as $context_{i3}$ into $context_{i31}$ and $context_{i32}$, taking into account parameter d.

In Figure 9, we illustrate $context_i$ which is split into $C_{i1}, C_{i2},$ and C_{i3} subcontexts and composed with model S. the exploration of $model \| C_{i1}$ successes (we note $\|$ as composition operator). The explorations of $model \| C_{i2}$ and $model \| C_{i3}$ fail. So, C_{i2} (resp., C_{i3}) is split into subcontexts C_{i21} and C_{i22} (C_{i31} and C_{i32} resp.,). In the same way, C_{i31} is split into subcontexts C_{i311}, C_{i312}, and C_{i313}. This algorithm is executed until all the explorations succeed. Since the property set pty is associated with the context $context_i$, pty is checked during the explorations with all subcontexts. We demonstrated in [31] that the verification of property set (as pty) taking into account of $model \| context_i$ exploration is equivalent union of verifications taking into account each $model \| C_k$ exploration (C_k is each subcontext $C_{i1}, C_{i21}, C_{i22}, C_{i32}, C_{i311}, C_{i312}, \text{and} C_{i313}$ as illustrated in Figure 9).

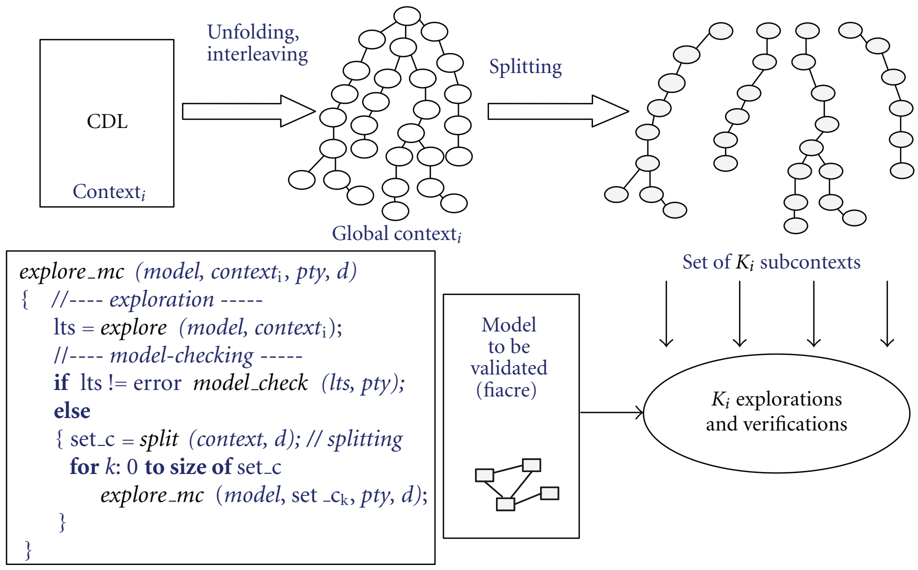
The following verification processes are then equivalent: (i) compose the context $context_i$ and the system, and then verify the resulting global system, (ii) partition the context $context_i$ into K_i subcontexts (scenarios), and successively deal each scenario with the model and check the properties on the outcome of each composition. Actually, we transform the global verification problem for $context_i$ into K_i smaller verification subproblems. In our approach, the complete context model can be split into pieces that have to be composed separately with the system model.

In summary, the context aware method provides three reduction axes: the context behavior is constrained, the properties are focused, and the state space is split into pieces. Finally, the N verifications for the set of N contexts is transformed into N' verifications with $N' = \sum_{i=1}^{N} K_i$ small verifications.

The reduction in the model behavior is particularly interesting while dealing with complex embedded systems, such as in avionic systems, since it is relevant to check properties over specific system modes (or use cases) which is less complex because we are dealing with a subset of the system automata. Unfortunately, only few existing approaches propose operational ways to precisely capture these contexts in order to reduce formal verification complexity and thus improve the scalability of existing model checking approaches. The necessity of a clear methodology has also to be identified, since the context partitioning is not trivial, that is, it requires the formalization of the context of the subset of functions under study. An associated methodology must be defined to help users for modeling contexts (out of scope of this paper).

4. CDL Language for Context and Property Specification

We propose a formal tool-supported framework that combines context description and model transformations to assist in the definition of requirements and of the environmental conditions in which they should be satisfied. Thus,



```
explore_mc (model, context_i, pty, d)
{   //---- exploration -----
    lts = explore (model, context_i);
    //---- model-checking -----
    if lts != error model_check (lts, pty);
    else
    { set_c = split (context, d); // splitting
      for k: 0 to size of set_c
          explore_mc (model, set_c_k, pty, d);
    }
}
```

FIGURE 7: Context splitting and verification for each partition (subcontext).

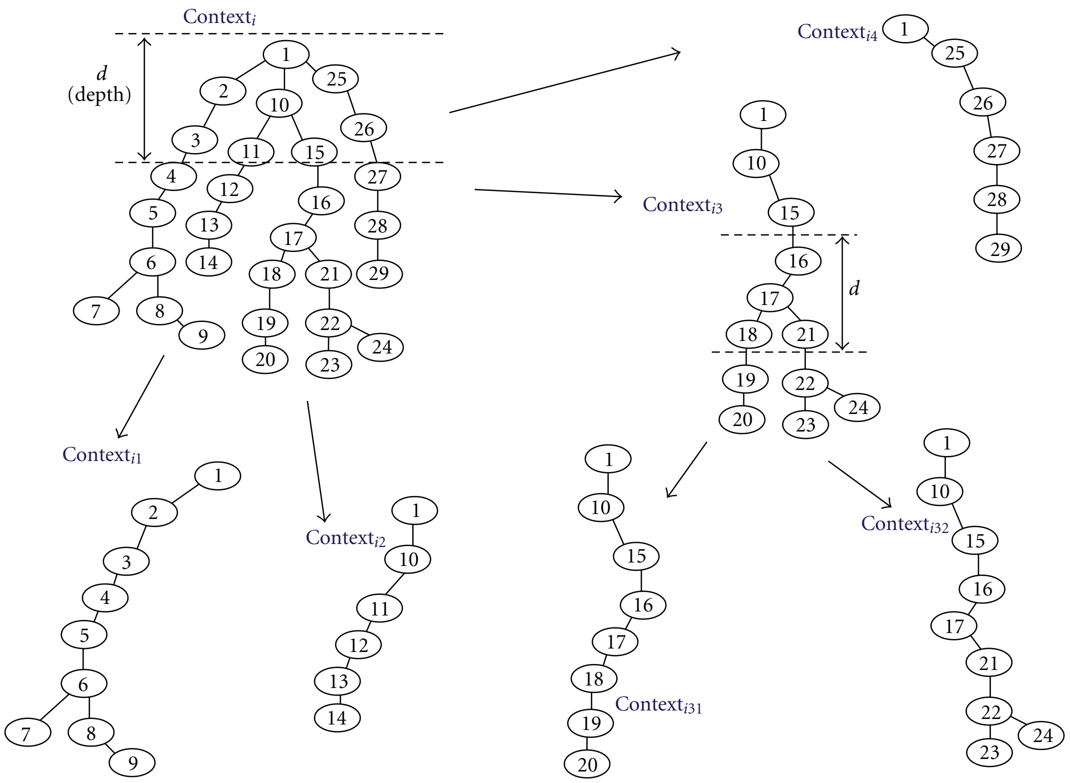


FIGURE 8: Context splitting algorithm.

we proposed [32] a context-aware verification process that makes use of the CDL language. CDL was proposed to fill the gap between user models and formal models required to perform formal verifications. CDL is a (domain specific language) presented either in the form of UML like graphical diagrams (a subset of activity and sequence diagrams) or in a textual form to capture environment interactions.

4.1. Context Hierarchical Description. CDL is based on Use Case Charts of [33] using activity and sequence diagrams. We extended this language to allow several entities (actors) to be described in a context (Figure 10). These entities run in parallel. A CDL (For the detailed syntax, see [34] available on http://www.obpcdl.org/.) model describes, on the one hand, the context using activity and sequence diagrams

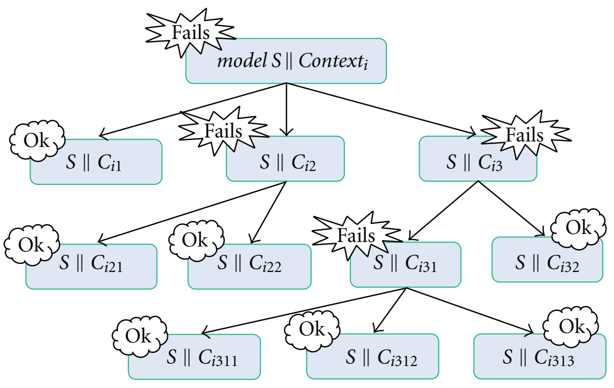


FIGURE 9: An example of context splitting execution.

and, on the other hand, the properties to be checked using property patterns. Figure 10 illustrates a CDL model for the partial use cases of Figures 1 and 2. Initial use cases and sequence diagrams are transformed and completed to create the context model. All context scenarios are represented, combined with parallel and alternative operators, in terms of CDL.

A diagrammatical and textual concrete syntax is created for the context description and a textual syntax for the property expression. CDL is hierarchically constructed in three levels: level 1 is a set of use case diagrams which describes hierarchical activity diagrams. Either alternative between several executions (alternative/merge) or a parallelization of several executions (fork/join) is available. Level 2 is a set of scenario diagrams organized in alternatives. Each scenario is fully described at level 3 by sequence diagrams. These diagrams are composed of lifelines, some for the context actors and others for processes composing the system model. Counters limit the iterations of diagram executions. This ensures the generation of finite context automata.

From a semantic point of view, we can consider that the model is structured in a set of sequence diagrams (MSCs) connected together with three operators: sequence (*seq*), parallel (*par*), and alternative (*alt*). The interleaving of context actors described by a set of MSCs generates a graph representing all executions of the actors of the environment. This graph is then partitioned in such a way as to generate a set of subgraphs corresponding to the subcontexts as mentioned in Section 3.3.

The originality of CDL is its ability to link each expressed property to a context diagram, that is, a limited scope of the system behavior. The properties can be specified with property pattern definitions described in [32, 34]. For checking, properties are linked to one or several context descriptions. Listing 3, we illustrate an example (textual version) of a scenario (*scenario_ex*) with linked properties: three observer-based properties $P1$, $P2$, and $P3$ (Pi ($i \in [1,\dots,3]$) property specifying requirement $R2$) and three invariants $Inv1$, $Inv2$, and $Inv1$ ($Invi$ ($i \in [1,\dots,3]$) property specifying requirement $R1$). As example, properties $P1$ and $Inv1$ are specified at Section 4.2.

The clause *init* specifies an initialization with an activity. Actors DEV1, DEV2, and DEV3 are specified with activities, by Listing 4.

In Listing 4, the operators “;” and “[]” are respectively the sequence (*seq*) and alternative (*alt*) operators. CDL is designed so that formal artifacts required by existing model checkers could be automatically generated from it. This generation is currently implemented in OBP described briefly in Section 5. The CDL formal syntax and semantics are presented in [35].

4.2. Property Specification Patterns. Property specifying needs to use powerful yet easy mechanisms for expressing temporal requirements of software source code. As example, requirements as $R1$ or $R2$ of the *S_CP* system described in Section 3.1 can refer to many events related to the execution of the model or environment. Also, a requirement can depends on an execution history that has to be taken into account as a constraint or precondition.

If we want to express these kinds of requirements with a temporal logic based language as LTL or CTL, the logical formulas are of great complexity and become difficult to read and to handle by engineers. So, for the property specification, we propose to reuse the categories of Dwyer patterns [26] and extend them to deal with more specific temporal properties which appear when high-level specifications are refined. Additionally, a textual syntax is proposed to formalize properties to be checked using property description patterns [28]. To improve the expressiveness of these patterns, we enriched them with options (*Prearity*, *Postarity*, *Immediacy*, *Precedence*, *Nullity*, and *Repeatability*) using annotations as [27]. Choosing among these options should help the user to consider the relevant alternatives and subtleties associated with the intended behavior. These annotations allow these details to be explicitly captured. During a future work, we will adapt these patterns taking into account the taxonomy of relevant properties, if this appears necessary.

We integrate property patterns description in the CDL language. Patterns are classified in families, which take into account the timed aspects of the properties to be specified. The identified patterns support properties of answer (*Response*), the necessity one (*Precedence*), of absence (*Absence*), of existence (*Existence*) to be expressed. The properties refer to detectable events like transmissions or receptions of signals, actions, and model state changes. The property must be taken into account either during the entire model execution, before, after, or between occurrences of events. Another extension of the patterns is the possibility of handling sets of events, ordered or not ordered similar to the proposal of [36]. The operators *AN* and *ALL*, respectively, specify if an event or all the events, ordered (Ordered) or not (Combined), of an event set are concerned with the property.

We illustrate these patterns with our case study. The given requirement $R2$ (Listing 2) must be interpreted and can be written with CDL in a property $P1$ as follow (cf. Listing 5). $P1$ is linked to the communication sequence

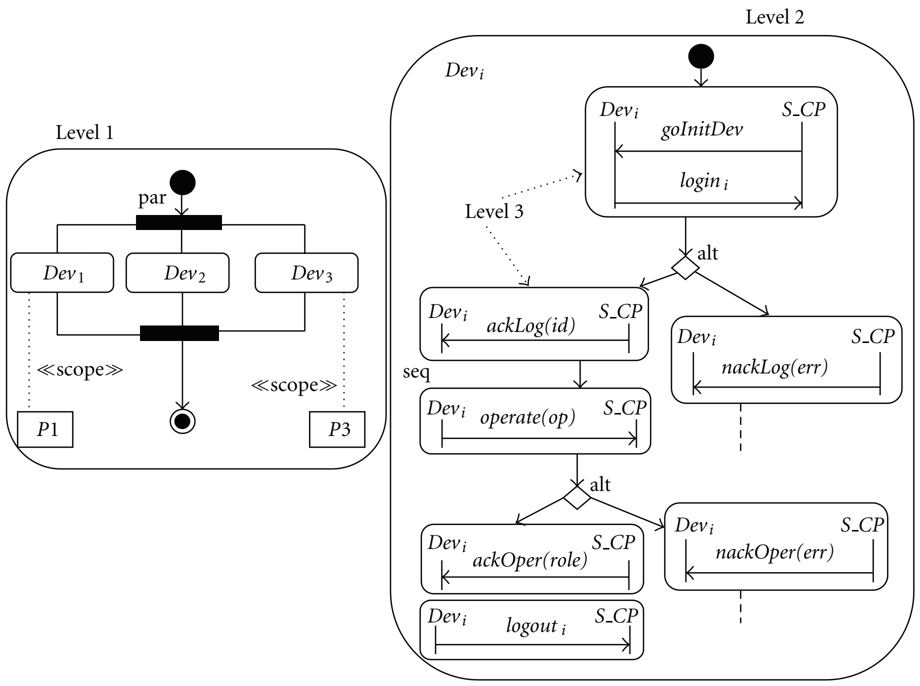


FIGURE 10: *S_CP* case study; partial representation of the context.

```
cdl scenario_ex is
{
 properties P1, P2, P3     //references to observers
 assert Inv1, Inv2, Inv3    //references to invariants
 init is { initDevs }                 //initialization sequence
 main is { DEV1 || DEV2 || DEV3 }        //body of scenario
}
```

LISTING 3: A CDL scenario with several observer-based properties and invariants.

between the *S_CP* and device (Dev_1). According to the sequence diagram of Figure 10, the association to other devices has no effect on *P*1.

*P*1 specifies an observation of event occurrences in accordance with Figure 10. *login*1 refers to $login_1$ reception event in the model, *ackLog* refers to ackLog reception event by Dev_1. *S_CP_hasReachState_Init* refers a state change in the model under study.

In CDL, we specify properties with events and predicates. For example, the event *S_CP_hasReachState_Init* is defined with predicate *S_CP_State_Init* as follows:

```
event S_CP_hasReachState_Init is
{S_CP_State_Init becomes true}
```

The predicate *S_CP_State_Init* is defined as follows:

```
predicate S_CP_State_Init is {{SM}
1@State_Init}
```

with *State_Init* as a state of process *SM*_1.

Invariants are specified with CDL predicats. As example, invariant *Inv*1 is specified as in Listing 6.

5. OBP Toolset

To carry out our experiments, we used OBP tool (Figure 11). OBP is an implementation of a CDL language translation in terms of formal languages, that is, currently Fiacre [30]. As depicted in Figure 11, OBP leverages existing academic model checkers such as TINA-SELT [3] or simulators such as OBP Explorer. From CDL context diagrams, the OBP tool generates a set of context graphs which represent the sets of the environment runs. Currently, each generated graph is transformed into a Fiacre automaton. Each graph represents a set of possible interactions between model and context. To validate the model under study, it is necessary to compose each graph with the model. Each property on

```
activity DEV1 is
{
   { event send_login1; {event recv_ack_log1 [] event recv_nack_log1}};
   { event send_operate1; {event recv_ack_oper1 [] event recv_nack_oper1}};
   { send logout1 to { SM }1}
}
```

Listing 4: An CDL activity.

```
Property P1;
    ALL Ordered
       exactly one occurrence of S_CP_hasReachState_Init
       exactly one occurrence of login1
    end
    eventually leads-to [0..maxD_log]
    AN
       one or more occurrence of ackLog (id)
    end
    S_CP_hasReachState_Init may never occurs
    login1 may never occurs
    one of ackLog (id) cannot occur before login1
    repeatability: true
```

Listing 5: A response pattern from $R2$ requirement.

each graph must be verified. In the case of TINA-SELT, the properties are expressed with SELT logic formula [3]. With OBP Explorer, OBP generates an observer automaton [37] from each property for OBP Explorer. With OBP Explorer, the accessibility analysis is carried out on the result of the composition between a graph, a set of observers and the system model as described in [32]. If for a given context, we face state explosion, the accessibility analysis, or model-checking is not possible. In this case, the context is split into a subset of contexts and the composition is executed again as mentioned in Section 3.3.

To import models with standard format such as UML, SysML, AADL, and SDL, we necessarily need to implement adequate translators such as those studied in Top-Cased (http://www.topcased.org/) or Omega (http://www-omega.imag.fr/.) projects to generate Fiacre programs.

6. Experiments and Results

Our approach was applied to several embedded systems applications in the avionic or electronic industrial domain. These experiments were carried out with our French industrial partners. In [32], we reported the results of these experiments. For the *S_CP* case study, we constructed several CDL models with different complexities depending on the number of devices. The tests are performed on each CDL model composed with *S_CP* system.

Table 3 shows the amount of TINA-SELT exploration and model checking (tests with same computer as for Table 1) for checking of requirement $R1$ with the use of context splitting. The first column depicts the number N of *Dev* asking for login to the *S_CP*. The third one indicates the number of subcontext after splitting by OBP. The other columns depict the exploration time and the cumulative amount of configurations and transitions of all LTS generated during exploration by TINA with context splitting. For example, with 7 devices, we needed to split the CDL context in 56 parts for successful exploration. Without splitting, the exploration is limited to 4 devices by state explosion as shown Table 1. It is clear that device number limit depends on the memory size of used computer.

Table 4 shows the amount of OBP Explorer exploration and analyze (tests with same computer as for Table 2) for checking of requirement $R2$ with the use of context splitting. With 7 devices, we needed to split the CDL context in 344 parts for successful exploration. Without splitting, the exploration is limited to three devices by state explosion as shown Table 2.

As mentioned previously in Section 3.2, the size of the LTS explored by OBP Explorer for verifying $R2$ is greater than the size of the related LTS explored by TINA-SELT for verifying $R1$. In that case, being able to split the contexts in order to overcome this new source of combinatorial explosion, as proposed by OBP, is of greater importance.

```
predicate Inv1 is
{ ({SM}1:operateAccepted1 = false) or ({SM}1:devLogged1 = true) }
```

Listing 6: A CDL invariant.

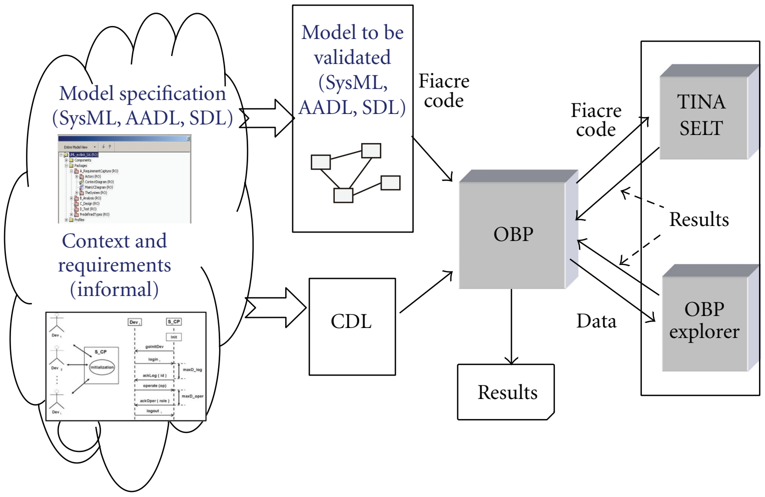

Model specification (SysML, AADL, SDL)
Context and requirements (informal)
Model to be validated (SysML, AADL, SDL)
Fiacre code
CDL
OBP
Fiacre code
TINA SELT
Results
OBP explorer
Data
Results

Figure 11: CDL model transformation with OBP.

The example given (Figure 1) illustrates a case where there are lots of asynchrony in the behavior of environment actors, causing an explosion in the number of states and thus an increase in the number of contextes generated. We obtain a good performance with this method in case the one hand, the contexts restrict significantly the behavior of the model to be validated (space-complexity reduction) and, secondly, in case the context number is not too large (time-complexity reduction).

Exploration can easily be parallelized. In fact, the splitting method allows contexts to be distributed on machine network. We do not yet implement this parallelization technique, but it can be very effective. Suppose we have a network of N similar machines. We can divide by N time of global exploration. This is an approximation by considering that the context transfer delays and result return delays are negligible compared to the exploration time on a machine. We should take into account the exploration time is not identical for all contexts. But parallelization can significantly improve the proof execution time. For example, in case shown in Table 4, with 20 machines (resp., 100), we can hope to obtain an execution time of approximately 5 minutes (resp. 1 minute) instead of two hours on a single machine. We believe our method of context splitting is complementary with other reduction methods. On some machine, for one subcontext, we can use another technique complementary way.

7. Discussion and Conclusion

CDL is a prototype language to formalize contexts and properties. However, CDL concepts can be implemented in another language. For example, context diagrams are easily described using full UML2. CDL permits us to study our methodology. In future work, CDL can be viewed as an intermediate language. Today, the results obtained using the currently implemented CDL language and OBP are very encouraging. For each case study, it was possible to build CDL models and to generate sets of context graphs with OBP.

During experiments, we noted that some contexts and requirements were often described in the available documentation in an incomplete way. During the collaboration with us, these engineers responsible for developing this documentation were motivated to consider a more formal approach to express their requirements, which is certainly a positive improvement.

In case studies, context diagrams were built, on the one hand, from scenarios described in the design documents and, on the other hand, from the sentences of requirement documents. Two major difficulties have arisen. The first is the lack of complete and coherent description of the environment behavior. Use cases describing interactions between the system (*S_CP* for instance) and its environment are often incomplete. For instance, data concerning interaction modes may be implicit. CDL diagram development thus requires discussions with experts who have designed the models under study in order to explicate all context assumptions. The problem comes from the difficulty in formalizing system requirements into formal properties. These requirements are expressed in several documents of different (possibly low) levels. Furthermore, they are written in a textual form, and many of them can have several interpretations. Others implicitly refer to an applicable configuration, operational

Table 3: Exploration and model checking of $R1$ with TINA-SELT with context splitting using OBP.

Number of devices	Exploration time (sec)	Number of sub-contexts	Number of LTS configurations	Number of LTS transitions
5	112	3	2,233,959	9,875,418
6	2,150	42	32,185,530	158,230,583
7	4,209	56	66,398,542	330,148,458

Table 4: Exploration and analyze of $R2$ with OBP Explorer with context splitting using OBP.

Number of devices	Exploration time (sec)	Number of sub-contexts	Number of LTS configurations	Number of LTS transitions
4	954	22	16,450,288	75,362,832
5	1,256	28	33,568,422	156,743,290
6	3,442	242	68,880,326	368,452,864
7	6,480	344	126,450,324	634,382,590

phase, or history without defining it. Such information, necessary for verification, can only be deduced by manually analyzing design and requirement documents and by interviewing expert engineers.

The use of CDL as a framework for formal and explicit context and requirement definition can overcome these two difficulties: it uses a specification style very close to UML and thus readable by engineers. In all case studies, the feedback from industrial collaborators indicates that CDL models enhance communication between developers with different levels of experience and backgrounds. Additionally, CDL models enable developers, guided by behavior CDL diagrams, to structure and formalize the environment description of their systems and their requirements.

One element highlighted when working on embedded software case studies with industrial partners, is the need for formal verification expertise capitalization. Given our experience in formal checking for validation activities it seems important to structure the approach and the data handled during the verifications. That can lead to a better methodological framework, and afterwards a better integration of validation techniques in model development processes. Consequently, the development process must include a step of environment specification making it possible to identify sets of bounded behaviors in a complete way.

Although the CDL approach has been shown scalable in several industrial case studies, the approach suffers from a lack of methodology. The handling of contexts, and then the formalization of CDL diagrams, must be done carefully in order to avoid combinatorial explosion when generating context graphs to be composed with the model to be validated. The definition of such a methodology will be addressed by the next step of this work.

References

[1] G. J. Holzmann, "The model checker SPIN," *IEEE Transactions on Software Engineering*, vol. 23, no. 5, pp. 279–295, 1997.

[2] K. G. Larsen, P. Pettersson, and W. Yi, "UPPAAL in a nutshell," *International Journal on Software Tools For Technology Transfer*, vol. 1, no. 1-2, pp. 134–152, 1997.

[3] B. Berthomieu, P. O. Ribet, and F. Vernadat, "The tool TINA—construction of abstract state spaces for petri nets and time petri nets," *International Journal of Production Research*, vol. 42, no. 14, pp. 2741–2756, 2004.

[4] J. C. Fernandez, H. Garavel, A. Kerbrat, L. Mounier, R. Mateescu, and M. Sighireanu, "Cadp: a protocol validation and verification toolbox," in *Proceedings of the 8th International Conference on Computer Aided Verification(CAV '96)*, pp. 437–440, London, UK, 1996.

[5] A. Cimatti, E. Clarke, F. Giunchiglia, and M. Roveri, "Nusmv: a new symbolic model checker," *The International Journal on Software Tools for Technology Transfer*, vol. 2, no. 4, pp. 410–425, 2000.

[6] E. M. Clarke, E. A. Emerson, and A. P. Sistla, "Automatic verification of finite-state concurrent systems using temporal logic specifications," *ACM Transactions on Programming Languages and Systems*, vol. 8, no. 2, pp. 244–263, 1986.

[7] G. J. Holzmann and D. Peled, "An improvement in formal verification," in *Proceedings of the Formal Description Techniques (FORTE '94)*, pp. 197–211, Chapman & Hall, Berne, Switzerland, October 1994.

[8] S. Park and G. Kwon, "Avoidance of state explosion using dependency analysis in model checking control flow model," in *Proceedings of the 5th International Conference on Computational Science and Its Applications (ICCSA '06)*, vol. 3984 of *Lecture Notes in Computer Science*, pp. 905–911, 2006.

[9] P. Godefroid, "The Ulg partial-order package for SPIN," in *Proceedings of the International Conference on Model Checking Software (SPIN '95)*, Montréal, Quebec, Canada, October 1995.

[10] D. Bošnački and G. J. Holzmann, "Improving Spin's partial-order reduction for breadth-first search," in *Proceedings of the 12th International SPIN Workshop on Model Checking Software (SPIN '05)*, pp. 91–105, August 2005.

[11] D. Peled, "Combining partial-order reductions with on-the-fly model-checking," in *Proceedings of the 6th International Conference on Computer Aided Verifiation (CAV '94)*, pp. 377–390, Springer, London, UK, 1994.

[12] A. Valmari, "Stubborn sets for reduced state space generation," in *Proceedings of the 10th International Conference on Applications and Theory of Petri Nets*, pp. 491–515, Springer, London, UK, 1991.

[13] J. R. Burch, E. M. Clarke, K. L. McMillan, D. L. Dill, and L. J. Hwang, "Symbolic model checking: 1020 states and beyond," in *Proceedings of the 5th Annual IEEE Symposium on Logic in Computer Science*, pp. 428–439, June 1990.

[14] E. M. Clarke, D. E. Long, and K. L. Mcmillan, *Compositional Model Checking*, MIT Press, 1999.

[15] C. Flanagan and S. Qadeer, “Thread-modular model checking,” in *Proceedings of the 10th International Conference on Model Checking Software (SPIN ’03)*, vol. 2648 of *Lecture Notes in Computer Science*, pp. 213–224, 2003.

[16] O. Tkachuk and M. B. Dwyer, “Automated environment generation for software model checking,” in *Proceedings of the 18th International Conference on Automated Software Engineering*, pp. 116–129, 2003.

[17] L. De Alfaro and T. A. Henzinger, “Interface automata,” in *Proceedings of the 8th Eiropean Engineering Conference and 9th ACM SIGSOFT Symposium on the Foundations of Software Engineering (FSE ’01)*, pp. 109–120, ACM Press, September 2001.

[18] J. Barnat, L. Brim, and J. Stříbrná, “Distributed ltl model-checking in spin,” in *Proceedings of the 8th International SPIN Workshop on Model Checking of Software (SPIN ’01)*, pp. 200–216, Springer, New York, NY, USA, 2001.

[19] S. Edelkamp and S. Schroedl, *Heuristic Search—Theory and Applications*, Academic Press, 2012.

[20] K. Havelund, A. Skou, K. G. Larsen, and K. Lund, “Formal modeling and analysis of an audio/video protocol: an industrial case study using UPPAAL,” in *Proceedings of the 18th IEEE Real-Time Systems Symposium*, pp. 2–13, December 1997.

[21] F. Boniol, V. Wiels, and E. Ledinot, “Experiences using model checking to verify real time properties of a landing gear control system,” in *Proceedings of the 5th European Congress ERTS Embedded Real Time Software (ERTS ’06)*, Toulouse, France, 2006.

[22] T. Bochot, P. Virelizier, H. Waeselynck, and V. Wiels, “Model checking flight control systems: the Airbus experience,” in *Proceedings of the 31st International Conference on Software Engineering (ICSE ’09)*, pp. 18–27, May 2009.

[23] X. Dumas, P. Dhaussy, F. Boniol, and E. Bonnafous, “Application of partial-order methods for the verification of closed-loop SDL systems,” in *Proceedings of the 26th Annual ACM Symposium on Applied Computing (SAC ’11)*, W. C. Chu, W. Eric Wong, M. J. Palakal, and C. C. Hung, Eds., pp. 1666–1673, March 2011.

[24] L. E. Moser, Y. S. Ramakrishna, G. Kutty, P. M. Melliar-Smith, and L. K. Dillon, “A graphical environment for the design of concurrent real-time systems,” *ACM Transactions on Software Engineering and Methodology*, vol. 6, no. 1, pp. 31–79, 1997.

[25] Y. S. Ramakrishna, L. K. Dillon, L. E. Moser, P. M. Melliar-Smith, and G. Kutty, “Real-time interval logic and its decision procedure,” in *Proceedings of the 13th Foundations of Software Technology and Theoretical Computer Science Conference (FSTTCS ’93)*, vol. 761 of *Lecture Notes in Computer Science*, pp. 173–192, 1993.

[26] M. B. Dwyer, G. S. Avrunin, and J. C. Corbett, “Patterns in property specifications for finite-state verification,” in *Proceedings of the 21st International Conference on Software Engineering*, pp. 411–420, IEEE Computer Society Press, May 1999.

[27] R. L. Smith, G. S. Avrunin, L. A. Clarke, and L. J. Osterweil, “PROPEL: an approach supporting property elucidation,” in *Proceedings of the 24th International Conference on Software Engineering (ICSE ’02)*, pp. 11–21, St Louis, Mo, USA, May 2002.

[28] S. Konrad and B. H. C. Cheng, “Real-time specification patterns,” in *Proceedings of the 27th International Conference on Software Engineering (ICSE ’05)*, pp. 372–381, St Louis, Mo, USA, May 2005.

[29] N. De Francesco, A. Santone, and G. Vaglini, “A user-friendly interface to specify temporal properties of concurrent systems,” *Information Sciences*, vol. 177, no. 1, pp. 299–311, 2007.

[30] P. Farail, P. Gaufillet, F. Peres et al., “FIACRE: an intermediate language for model verification in the TOPCASED environment,” in *Proceedings of the European Congress on Embedded Real-Time Software (ERTS ’08)*, Toulouse, France, Janvier 2008.

[31] J. C. Roger, *Exploitation de contextes et d’observateurs pour la validation formelle de modèles [Ph.D. thesis]*, ENSIETA, Université de Rennes I., December 2006.

[32] P. Dhaussy, P. Y. Pillain, S. Creff, A. Raji, Y. Le Traon, and B. Baudry, “Evaluating context descriptions and property definition patterns for software formal validation,” in *Proceedings of the 12th IEEE/ACM Conference Model Driven Engineering Languages and Systems (Models ’09)*, vol. 5795 of *Lecture Notes in Computer Science*, pp. 438–452, Springer, 2009.

[33] J. Whittle, “Specifying precise use cases with use case charts,” in *Proceedings of the International Conference on Satellite Events at the MoDELS (MoDELS ’06)*, vol. 3844 of *Lecture Notes in Computer Science*, pp. 290–301, 2006.

[34] P. Dhaussy and J. C. Roger, “Cdl (context description language): syntax and semantics,” Tech. Rep., ENSTA, Bretagne, France, 2011.

[35] P. Dhaussy, F. Boniol, and J.-C. Roger, “Reducing state explosion with context modeling for model-checking,” in *Proceedings of the 13th IEEE International High Assurance Systems Engineering Symposium (Hase ’11)*, Boca Raton, Fla, USA, 2011.

[36] W. Janssen, R. Mateescu, S. Mauw, P. Fennema, and P. Van Der Stappen, “Model checking for managers,” in *Proceedings of the International SPIN Workshop on Model Checking of Software*, pp. 92–107, 1992.

[37] N. Halbwachs, F. Lagnier, and P. Raymond, “Synchronous observers and the verification of reactive systems,” in *Proceedings of the 3rd International Conference on Algebraic Methodology and Software Technology (AMAST ’93)*, M. Nivat, C. Rattray, T. Rus, and G. Scollo, Eds., Workshops in Computing, Springer, Twente, The Netherlands, June 1993.