Context Aware Model Exploration with OBP tool to Improve Model-Checking

Philippe Dhaussy, Jean-Charles Roger, Luka Leroux
LabSticc ENSTA Bretagne
Brest France
Email: name.firstname@ensta-bretagne.fr

Frédéric Boniol
ONERA/DTIM
Toulouse France
Email: boniol@onera.fr

Abstract—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. This article shows how this combinatorial explosion could be reduced by specifying the environment of the system to be validated. Our contribution is illustrated on an industrial embedded system and using the TINA and OBP model-checkers.

Keywords—model-checking; use cases; context;

I. 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 [Hol97], [LPY97], [BRV04], [FGK+96], [CCGR00] 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 [CES86], [HP94], [PK06]. 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 article presents an exploratory work to provide solutions to the problems mentioned above. It is based on two joint ideas: first, to reduce behaviors system to be validated during model-checking and secondly, help the user to specify the formal properties to check. 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. Each context corresponds to an operational phase identified as system initialization, reconfiguration, graceful degradation, etc.. In addition, each context is associated with a set of properties to check. The aim is to guide the model-checker to on a restriction of the system behavior for verification of specific properties instead on exploring the global automaton.

In this paper, we describe the formalism, such as DSL1 called CDL (Context Description Language), useful for describing the contexts. This language serves to support our approach to reduce the state space. We illustrate our reduction technique with our OBP (Observer Based Prover)2 tool connected to two tools: The first is an academic model-checker TINA-SELT3 [BRV04] 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 II presents the related techniques to improve model checking by state reduction. Section III presents the principles of our approach for context aware formal verification. Section IV describes the CDL language for contexts specification and property specification. Our toolset used for the experiments is presented section V. In Section VI, we give results on the industrial case study. In section VII, we discuss our approach and we conclude.

1Domain Specific Language
2OBP is available on http://www.obpcdl.org.
II. RELATED WORKS

Several model checkers have been developed to help the verification of concurrent asynchronous systems. For example, the SPIN model-checker [Hol97] based on the formal language PROMELA allows the verification of LTL properties encoded in "never claim" formalism and further converted into Buchi automata. Several techniques have been investigated in order to improve the performance of SPIN. For instance the state compression method or partial-order reduction contributed to the further alleviation of combinatorial explosion [God95]. In [BH05] 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 [Pel94] [Val91] [God95] 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).

Compositional (modular) specification and analysis techniques have been researched for a long time and resulted in, e.g., assume/guarantee reasoning or design-by-contract techniques. A lot of work exists in applying these techniques to model checking including, e.g. [CLM99], [FQ03], [TD03], [AH01] 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.

Our approach is different from compositional or modular analysis. Context aware verification is not about verifying component by component. With the "traditional model checking", 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. However, our approach can used in conjunction with design by contract process.

Another difficulty is about requirement specification. Embedded software systems integrate more and more advanced features, such as complex data structures, recursion, 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 [CES86]) 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, i.e., they are designed having in mind the straightforwardness of its processing by a tool such as a model-checker rather than the user-friendliness. Their concrete syntax is often simplistic, tailored for easing its processing by particular tools such as model checkers. 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.

It is thus necessary to facilitate the requirement expression with adequate languages by abstracting some details in the property description, at a price of reducing the expressivity. This conclusion was drawn a long time ago and several researchers [DAC99], [SAC02], [KC05] proposed to formulate the properties using definition patterns in order to assist engineers in expressing system requirements. Patterns are 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.

III. 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-SLT model checker [BRV04], and OBP Explorer. We present the results for a part of the \( S_CP \) model. Then, we introduce our approach based on context specifications.

A. An illustration

We present one part of an industrial case study: the software part of an anti-aircraft 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 interacts with devices (\( \text{Dev} \)) that are considered to be \( \text{actors} \) included in the \( S_CP \) environment called here \( \text{context} \).

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 \( \text{Dev} \) running in parallel or in sequence. All these actors interleave their behavior. After the initializing phase, all actors \( \text{Dev}_i \) \((i \in [1 \ldots n])\) wait for orders \( \text{goInitDev} \) from the system. Then, actors \( \text{Dev}_i \) send \( \text{login}_i \) and receive either \( \text{ackLog(id)} \) (Figure 2.a and 2.c) or \( \text{nackLog(err)} \) (Figure 2.b) as responses from the system. The logged devices can send \( \text{operate(op)} \) (Figure 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 Listing 1 and Listing 2.

The first requirement R1 (Listing 1) is expressed by:

**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”.

We choose to specify this requirement with SELT language for the device Dev_1. It is expressed by the following formula:

\[ \{ (SM_1_voperateAccepted1) \Rightarrow (SM_1_vdevLogged1) \}; \]

SM_1 is a process of S_CP and operateAccepted1 and devLogged1 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 R2.

**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.

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 S and its context C (and thus events occurring in the executions (runs)) and which produces an event reject whenever the 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\(^4\) is translated into Fiacre format [FGP+08] to explore all the S_CP model behaviors by simulation, S_CP interacting with its environment.

\(^4\)Here by system model, we refer to the model to be validated.
Model exploration generates a labeled transition system (LTS) which represents all the behaviors of the controller in its environment.

**B. Model-checking results**

Table I shows 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>
<thead>
<tr>
<th>$N$ (Number of devices)</th>
<th>Exploration &amp; model-checking time (sec)</th>
<th>N.of LTS configurations</th>
<th>N.of LTS transitions</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>1</td>
<td>22 977</td>
<td>103 354</td>
</tr>
<tr>
<td>2</td>
<td>3</td>
<td>172 095</td>
<td>759 094</td>
</tr>
<tr>
<td>3</td>
<td>10</td>
<td>718 623</td>
<td>3 127 468</td>
</tr>
<tr>
<td>4</td>
<td>27</td>
<td>2 174 997</td>
<td>9 371 560</td>
</tr>
<tr>
<td>5</td>
<td>Explosion</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Table II shows the OBP Explorer exploration-analyze time and the amount of configurations and transitions in the LTS. Over three devices, we see also a state explosion because of the limited memory of our computer.

<table>
<thead>
<tr>
<th>$N$ (Number of devices)</th>
<th>Exploration &amp; analyze time (sec)</th>
<th>N.of LTS configurations</th>
<th>N.of LTS transitions</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>1</td>
<td>43 828</td>
<td>321 002</td>
</tr>
<tr>
<td>2</td>
<td>4</td>
<td>350 256</td>
<td>2 475 392</td>
</tr>
<tr>
<td>3</td>
<td>19</td>
<td>1 466 934</td>
<td>6 430 265</td>
</tr>
<tr>
<td>4</td>
<td>Explosion</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

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.

**C. 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 technics 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 ($context_i$, $i \in [1..N]$ 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 sub-contexts. Here we precisely define these two aspects implemented in our approach.

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.

In this approach, we suppose that the designer is able to identify all possible interactions between the system and its environment. We also consider that each context

![Figure 5. Verification with OBP Explorer.](image)

![Figure 6. Context-aware model checking.](image)

---

5Tests were executed on Linux 32 bits - 3 Go RAM computer, with TINA vers.2.9.8 and Frac parser vers.1.6.2.

6Tests were executed on Linux 32 bits - 3 Go RAM computer, with OBP Explorer vers.1.0.
expressed initially is finite, (i.e., there is a non infinite loop in the context). We justify this strong hypothesis, particularly in the field of embedded systems, by 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. It would be necessary to study formally the validity of this 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, 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.

The second idea is to automatically split each identified context into a set of smaller sub-contexts (Figure 7). The following verification process is 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 $K_i$ sub-contexts (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 sub problems. In our approach, the complete context model can be split into pieces that have to be composed separately with the system model. To reach that goal, we implemented a recursive splitting algorithm in our OBP tool. Figure 7 illustrates the function $explore_me()$ for exploration of a model, with a $context_i$ and model-checking of a set of properties $ply$. The context is represented by acyclic graph. 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 $d$ for the depth in the graph for splitting) until the exploration succeeds.

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, i.e., 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).

IV. 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, we proposed [DPC+09] 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.

A. Context hierarchical description

CDL is based on Use Case Charts of [Whi06] using activity and sequence diagrams. We extended this language to allow several entities (actors) to be described in a context (Figure 8). These entities run in parallel. A CDL$^3$ model describes, on the one hand, the context using activity and sequence diagrams and, on the other hand, the properties to be checked using property patterns. Figure 8 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

\$^3\$For the detailed syntax, see [DR11] available on www.obpcdl.org.
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 sub-contexts as mentioned in III-C.

\[ S_{CP} \] case study: partial representation of the context

The originality of CDL is its ability to link each expressed property to a context diagram, i.e. a limited scope of the system behavior. The properties can be specified with property pattern definitions described in [DPC+09] and [DR11]. Properties can be linked to the context description at Level 1 or Level 2 (such as \( P_1 \) and \( P_3 \) in Figure 8) by the stereotyped links property/scope. A property can have several scopes and several properties can refer to a single diagram. 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 V. The CDL formal syntax and semantics are presented in [DBR11].

\section*{B. Property specification patterns}

Property specifying needs to use powerful yet easy mechanisms for expressing temporal requirements of software source code. As example, requirements as \( R_1 \) or \( R_2 \) of the \( S_{CP} \) system described in section III-A 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 pre-condition.

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 [DAC99] 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 [KC05]. To improve the expressiveness of these patterns, we enriched them with options (Pre-arity, Post-arity, Immediacy, Precedence, Nullity, Repeatability) using annotations as [SACO02]. 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 [JMM*+99]. 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 \( R_2 \) (Listing 2) must be interpreted and can be written with CDL in a property \( P \) as follow (cf. Listing 2). \( P \) is linked to the communication sequence between the \( S_{CP} \) and device (\( Dev_1 \)). According to the sequence diagram of figure 8, the association to other devices has no effect on \( P \).

\begin{verbatim}
Property P;
  ALL Ordered
    exactly one occurrence of S_CP_hasReachState_Init
  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
\end{verbatim}
one of ackLog (id) cannot occur before login1
repeatability: true

Listing 3. A response pattern from R2 requirement.
P specifies an observation of event occurrences in accordance with figure 8. login1 refers to login1 reception event in the model, ackLog refers to ackLog reception event by Dev1. 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:

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

The predicate S_CP_State_Init is defined as:

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

with State_Init as a state of process SM_1.

V. OBP TOOLSET

To carry out our experiments, we used OBP tool (Figure 9). OBP is an implementation of a CDL language translation in terms of formal languages, i.e. currently Fiacre [FGP+08]. As depicted in Figure 9, OBP leverages existing academic model checkers such as TINA-SELT [BRV04] 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 each graph must be verified. In the case of TINA-SELT, the properties are expressed with SELT logic formula [BRV04]. With OBP Explorer, OBP generates an observer automaton [HLR93] 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 [DPC+09]. 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 III-C.

To import models with standard format such as UML, SysML, AADL, SDL, we necessarily need to implement adequate translators such as those studied in TopCased8 or Omega9 projects to generate Fiacre programs.

VI. 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 [DPC+09], 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 III shows the amount of TINA-SELT exploration and model-checking10 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 second one indicates le number of sub-context 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 I. It is clear that device number limit depends on the memory size of used computer.

<table>
<thead>
<tr>
<th>N. of devices</th>
<th>Exploration time (sec)</th>
<th>N.of sub-contexts</th>
<th>N. of LTS config.</th>
<th>N. of LTS trans.</th>
</tr>
</thead>
<tbody>
<tr>
<td>5</td>
<td>112</td>
<td>3</td>
<td>2 233 959</td>
<td>9 875 418</td>
</tr>
<tr>
<td>6</td>
<td>2 150</td>
<td>42</td>
<td>32 185 530</td>
<td>158 230 583</td>
</tr>
<tr>
<td>7</td>
<td>4 209</td>
<td>56</td>
<td>66 398 542</td>
<td>330 148 458</td>
</tr>
</tbody>
</table>

Table IV shows the amount of OBP Explorer exploration and analyze11 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 II.

As mentioned previously in section III-B, the size of the LTS explored by OBP Explorer for verifying R2 is greater

8www.topcased.org
9www-Omega.imag.fr
10Tests with same computer as for Table I
11Tests with same computer as for Table II
Table IV

<table>
<thead>
<tr>
<th>N. of devices</th>
<th>Exploration time (sec)</th>
<th>N. of sub-contexts</th>
<th>N. of LTS config.</th>
<th>N. of LTS trans.</th>
</tr>
</thead>
<tbody>
<tr>
<td>4</td>
<td>954</td>
<td>22</td>
<td>16 450 288</td>
<td>75 562 832</td>
</tr>
<tr>
<td>5</td>
<td>1 256</td>
<td>28</td>
<td>33 568 422</td>
<td>156 743 290</td>
</tr>
<tr>
<td>6</td>
<td>3 442</td>
<td>242</td>
<td>68 880 326</td>
<td>568 452 864</td>
</tr>
<tr>
<td>7</td>
<td>6 480</td>
<td>344</td>
<td>126 450 324</td>
<td>634 382 590</td>
</tr>
</tbody>
</table>

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


