Reducing State Explosion with Context Modeling for Model-Checking

Philippe Dhaussy, Jean-Charles Roger
LISyC 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.**

***Keywords*-model-checking; use cases; context;**

I. Introduction

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], 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 focus instead on exploring the global automaton but a restriction of the latter relevant for verification of specific properties.

In this paper, we describe the formalism, such as DSL[1] called CDL (Context Description Language). This language serves to support our approach to reduce the state space. We report a feedback on several case studies industrial field of aeronautics, which was conducted in close collaboration with engineers in the field.

This paper is organized as follows: Section II presents the 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. Section V presents our tool used for the experiments. In Section VI we give the selected results of an industrial case study. Section VII discusses our approach and presents future work.

II. Related works

Several model checkers such as SPIN [Hol97], Uppaal [LPY97], TINA [BRV04], have been developed to help the verification of concurrent asynchronous systems. For example, the SPIN model-checker 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

[1]Domain Specific Language

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.

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-SELT model checker [BRV04]. 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 (Dev) that are considered to be $actors$ included in the S_CP environment called here $context$.

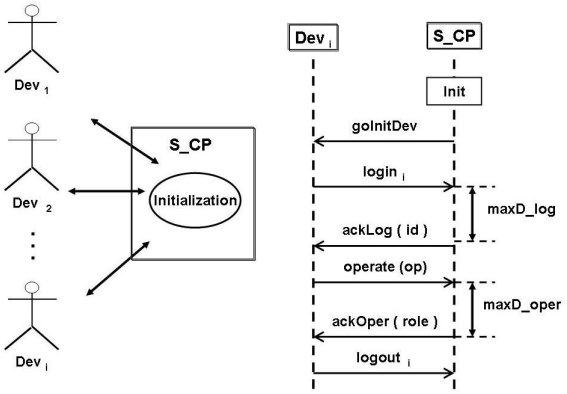

Figure 1. S_CP system: partial description during the initialization phase

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 verification 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 ($i \in [1 \dots n]$) wait for orders $goInitDev$ from the system. Then, actors Dev_i send $login_i$ and receive either $ackLog(id)$ (Figure 2.a and 2.c) or $nackLog(err)$ (Figure 2.b) as responses from the system. The logged devices can send $operate(op)$ (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.

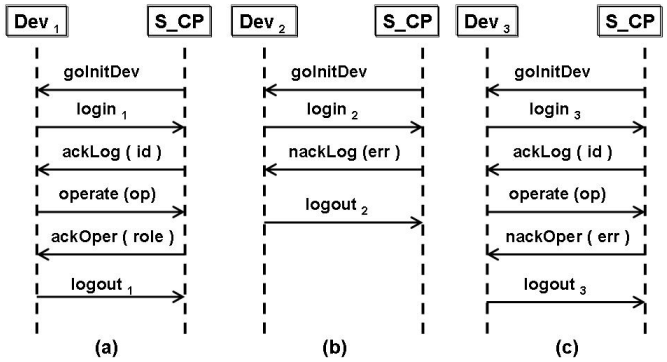

Figure 2. An example of S_CP context scenario with 3 devices.

B. Model-checking results

To verify requirements on the system model[2], we used the TINA-SELT model checker [BRV04]. To do so, the system model is translated into FIACRE format [FGP+08] 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. Table I shows[3] the 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 I
TABLE HIGHLIGHTING THE VERIFICATION COMPLEXITY FOR AN INDUSTRIAL CASE STUDY.

N.of devices	Exploration time (sec)	N.of LTS configurations	N.of LTS transitions
1	10	16 766	82 541
2	25	66 137	320 388
3	91	269 977	1 297 987
4	118	939 689	4 506 637
5	Explosion	–	–

C. Combinatorial explosion reduction

When checking the properties of a model, 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 (Figure 3.a). 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 3.b). 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 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.

[2]Here by system or system model, we refer to the model to be validated.
[3]Tests were executed on Linux 32 bits - 3 Go RAM computer, with TINA vers.2.9.8 and Frac parser vers.1.4.2.

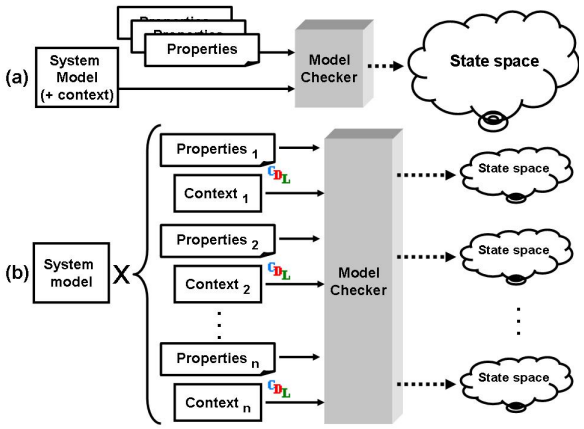

Figure 3. Traditional model checking (a) vs. context-aware model checking (b)

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 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 4). The following verification process is then equivalent: (i) compose the context and the system, and then verify the resulting global system, (ii) partition the environment into k 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 into k 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 4) illustrates the function $explore\_mc()$ for exploration of a $model$, with a $context$ and model-checking of a set of properties pty. 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.

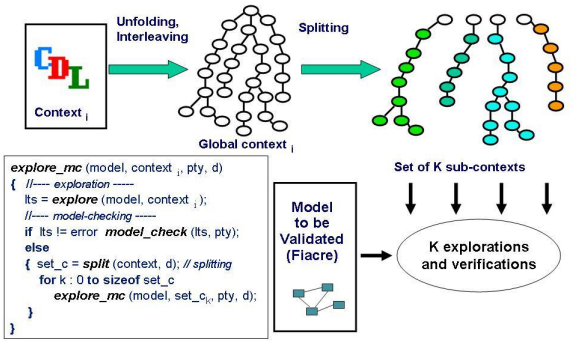

Figure 4. Context splitting and verification for each partition (sub-context).

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. 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 5). These entities run in parallel. A CDL[4] 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 5 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 sub-contexts as mentioned in III-C.

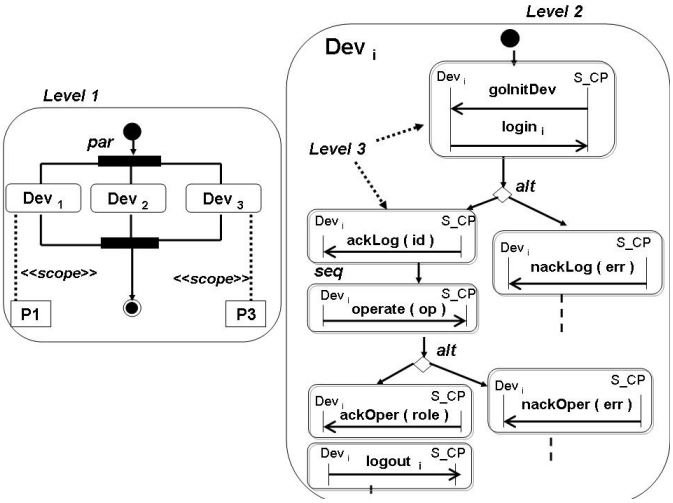

Figure 5. S_CP case study: partial representation of the context

The originality of CDL is its ability to link each expressed property to a context diagram, i.e. a limited scope of the sys-

[4]For the detailed syntax, see [DR11] available (currently in french) on www.obpcdl.org.

tem behavior. The properties can be specified with property pattern definitions that we do not describe here but can be found in [DPC+09] and [DR11]. Properties can be linked to the context description at Level 1 or Level 2 (such as $P1$ and $P3$ in Figure 5) 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 our prototype tool called OBP (*Observer Based Prover*) described briefly in Section V. We will now present the CDL formal syntax and semantics.

B. Formal Syntax

A CDL model (also called "context") is a finite generalized MSC C, following the formal grammar:

$$\begin{array}{lll} C & ::= & M \mid C_1; C_2 \mid C_1 + C_2 \mid C_1 \| C_2 \\ M & ::= & \mathbf{0} \mid a!; M \mid a?; M \end{array}$$

In other words, a context is either (1) a single MSC M composed as a sequence of event emissions $a!$ and event receptions $a?$ terminated by the empty MSC ($\mathbf{0}$) which does nothing, or (2) a sequential composition (seq denoted ;) of two contexts ($C_1; C_2$), or (3) a non deterministic choice (alt denoted +) between two contexts ($C_1 + C_2$), or (4) a parallel composition (par denoted $\|$) between two contexts ($C_1 \| C_2$).

For instance, let us consider the context Figure 5 graphically described. This context describes the environment we want to consider for the validation of the system model. We consider that the environment is composed of 3 actors Dev_1, Dev_2 and Dev_3. All these actors run in parallel and interleave their behavior. The model can be formalized, with the above textual grammar as follows[5].

$$\begin{array}{l} C \quad = Dev_1 \parallel Dev_2 \parallel Dev_2 \\ Dev_i = Log_i; (Oper + (nackLog\ (err)?;\ \ldots\ .\mathbf{0})) \\ Log_i \ = (goInitDev\ ?\ ;\ login_i\ !) \\ Oper = (\ ackLog\ (id)\ ?\ ;\ operate\ (op)\ !\ (\ Ack_i \\ \qquad + (\ nackOper\ (err)\ ?\ ;\ \ldots\ ; \mathbf{0}))) \\ Ack_i = (\ ackOper\ (role)\ ?\ ;\ logout_i\ !\ ;\ \ldots ; \mathbf{0}) \\ Dev_1, Dev_2,\ Dev_3\ =\ Dev_i\ with\ i\ =\ 1,\ 2,\ 3 \end{array}$$

C. Semantics

The semantics is based on the semantics of the scenarios and expressed by construction rules of sets of traces built using seq, alt and par operators. A scenario trace is an ordered events sequence which describes a history of the interactions between the context and the model.

To describe the formal semantics, let us define a function *wait*(C) associating the context C with the set of events awaited in its initial state:

$$\begin{array}{l} \textit{Wait}\ (\mathbf{0}) \stackrel{\text{def}}{=} \emptyset \qquad \textit{Wait}\ (a!; M) \stackrel{\text{def}}{=} \emptyset \\ \textit{Wait}\ (a?; M) \stackrel{\text{def}}{=} \{a\} \\ \textit{Wait}\ (C_1 + C_2) \stackrel{\text{def}}{=} \textit{Wait}\ (C_1) \cup \textit{Wait}\ (C_2) \\ \textit{Wait}\ (C_1; C_2) \stackrel{\text{def}}{=} \textit{Wait}\ (C_1)\ si\ C_1\ \neq\ \mathbf{0} \\ \textit{Wait}\ (\mathbf{0}; C_2) \stackrel{\text{def}}{=} \textit{Wait}\ (C_2) \\ \textit{Wait}\ (C_1 \| C_2) \stackrel{\text{def}}{=} \textit{Wait}\ (C_1) \cup \textit{Wait}\ (C_2) \end{array}$$

We consider that a context is a process communicating in an asynchronous way with the system, memorizing its input events (from the system) in a *buffer*. The semantics of CDL is defined by the relation $(C, B) \xrightarrow{a} (C', B')$ to express that the context C with the buffer B "produces" a (which can be a sending or a receiving signal, or the $null_\sigma$ signal if C does not evolve) and then becomes the new context C' with the new buffer B'. This relation is defined by the 8 rules in Figure 6 (In these rules, a represents an event which is different from $null_\sigma$):

- The *pref1* rule (without any preconditions) specifies that an MSC beginning with a sending event $a!$ emits this event and continues with the remaining MSC.
- The *pref2* rule expresses that if an MSC begins by a reception a? and faces an input buffer containing this event at the head of the buffer, the MSC consumes this event and continues with the remaining MSC.
- The *seq1* rule establishes that a sequence of contexts $C_1; C_2$ behaves as C_1 until it has terminated. The *seq2* rule says that if the first context C_1 terminates (i.e., becomes $\mathbf{0}$), then the sequence becomes C_2.
- The *par1* and *par2* rules say that the semantics of the parallel operation is based on an asynchronous interleaving semantics.
- The *alt* rule expresses that the alternative context $C_1 + C_2$ behaves either as C_1 or as C_2.
- Finally, the *discard* rule says that if an event a at the head of the input buffer is not expected, then this event is lost (removed from the head of the buffer).

D. Context and system composition

We can now formally define the "closure" composition $< (C, B_1) \mid (s, \mathcal{S}, B_2) >$ of a system $\mathcal{S}$ in a state $s \in \Sigma$ (Σ is the set of system states), with its input buffer B_2, with its context C, with its input buffer B_1 (note that each component, system and context, has its own buffer). The evolution of $\mathcal{S}$ closed by C is given by two relations: the relation (1):

$$< (C, B_1) | (s, \mathcal{S}, B_2) > \ \xrightarrow[\sigma]{a}\ < (C', B_1') | (s', \mathcal{S}, B_2') > \quad (1)$$

to express that $\mathcal{S}$ in the state s evolves to state s' receiving event a, potentially empty ($null_e$), (sent by the context) and producing the sequence of events σ, potentially empty ($null_\sigma$) (to the context). and the relation (2):

[5]In this paper, as an illustration, we consider that the behavior of actors extends, noted by the "..."

$$\frac{}{(a!;M,B) \xrightarrow{a!} (M,B)} \text{[pref1]}$$

$$\frac{}{(a?;M,a.B) \xrightarrow{a?} (M,B)} \text{[pref2]}$$

$$\frac{C_1' \neq \mathbf{0} \quad (C_1,B) \xrightarrow{a} (C_1',B')}{(C_1.C_2,B) \xrightarrow{a} (C_1'.C_2,B')} \text{[seq1]}$$

$$\frac{(C_1,B) \xrightarrow{a} (\mathbf{0},B')}{(C_1.C_2,B) \xrightarrow{a} (C_2,B')} \text{[seq2]}$$

$$\frac{C_1' \neq \mathbf{0} \quad (C_1,B) \xrightarrow{a} (C_1',B')}{\begin{array}{l}(C_1\|C_2,B) \xrightarrow{a} (C_1'\|C_2,B') \\ (C_2\|C_1,B) \xrightarrow{a} (C_2\|C_1',B')\end{array}} \text{[par1]}$$

$$\frac{(C_1,B) \xrightarrow{a} (\mathbf{0},B')}{\begin{array}{l}(C_1\|C_2,B) \xrightarrow{a} (C_2,B') \\ (C_2\|C_1,B) \xrightarrow{a} (C_2,B')\end{array}} \text{[par2]}$$

$$\frac{(C_1,B) \xrightarrow{a} (C_1',B')}{\begin{array}{l}(C_1+C_2,B) \xrightarrow{a} (C_1',B') \\ (C_2+C_1,B) \xrightarrow{a} (C_1',B')\end{array}} \text{[alt]}$$

$$\frac{a \notin \mathit{wait}(C)}{(C,a.B) \xrightarrow{\mathit{null}_\sigma} (C,B)} \text{[discard}_C\text{]}$$

Figure 6. Context semantics

$$< (C,B_1)|(s,\mathcal{S},B_2) > \xrightarrow[\sigma]{t} < (C,B_1)|(s',\mathcal{S},B_2') > \quad (2)$$

to express that $\mathcal{S}$ in state s evolves to the state s' by progressing time t, and producing the sequence of events σ potentially empty (null_σ) (to the context). Note that in the case of timed evolution, only the system evolves, the context is not timed. The semantics of this composition is defined by the four following rules (Figure 7):

- rule *cp1*: If $\mathcal{S}$ can produce σ, then $\mathcal{S}$ evolves and σ is put at the end of the buffer of C.
- rule *cp2*: If C can emit a, C evolves and a is queued in the buffer of $\mathcal{S}$.
- rule *cp3*: If C can consume a, then it evolves whereas $\mathcal{S}$ remains the same.
- rule *cp4*:If the time can progress in $\mathcal{S}$, then the time progress in the composition $\mathcal{S}$ and C.

Note that the "closure" composition between a system and its context can be compared with an asynchronous parallel composition: the behavior of C and of $\mathcal{S}$ are interleaved, and they communicate through asynchronous buffers. We will denote $< (C,B)|(s,\mathcal{S},B') > \not\rightarrow$ to express that the system and its context cannot evolve (the system is blocked

[cp1]
$$\frac{(s,\mathcal{S},B_2) \overrightarrow{\sigma} (s',\mathcal{S},B_2')}{< (C,B_1)|(s,\mathcal{S},B_2) > \xrightarrow[\sigma]{\mathit{null}_e} < (C,B_1.\sigma)|(s',\mathcal{S},B_2') >}$$

[cp2]
$$\frac{(C,B_1) \xrightarrow{a!} (C',B_1')}{< (C,B_1)|(s,\mathcal{S},B_2) > \xrightarrow[\mathit{null}_\sigma]{a} < (C',B_1')|(s,\mathcal{S},B_2.a) >}$$

[cp3]
$$\frac{(C,B_1) \xrightarrow{a?} (C',B_1')}{< (C,B_1)|(s,\mathcal{S},B_2) > \xrightarrow[\mathit{null}_\sigma]{\mathit{null}_e} < (C',B_1')|(s,\mathcal{S},B_2) >}$$

[cp4]
$$\frac{(s,\mathcal{S},B_2) \overrightarrow{\sigma} (s',\mathcal{S},B_2')}{< (C,B_1)|(s,\mathcal{S},B_2) > \xrightarrow[\sigma]{t} < (C,B_1)|(s',\mathcal{S},B_2') >}$$

Figure 7. CDL context and system composition semantics

or the context terminated). We then define the set of traces (called *runs*) of the system closed by its context from a state s, by:

$$\begin{aligned}[\![C \mid (s,\mathcal{S})]\!] \stackrel{\text{def}}{=} \{ & a_1 \cdot \sigma_1 \cdot \ \ldots \ a_n \cdot \sigma_n \cdot \mathit{end}_C \mid \\ & < (C,\mathit{null}_\sigma) \mid (s,\mathit{null}_\sigma) > \xrightarrow[\sigma_1]{a_1} \\ & < (C_1,B_1) \mid (s_1,\mathcal{S},B_1') > \xrightarrow[\sigma_2]{a_2} \ \ldots \ \xrightarrow[\sigma_n]{a_n} \\ & < (C_n,B_n) \mid (s_n,\mathcal{S},B_n') > \ \not\rightarrow \ \}\end{aligned}$$

$[\![C|(s,\mathcal{S})]\!]$ is the set runs of $\mathcal{S}$ closed by C from the state s. Note that a context is built as sequential or parallel compositions of finite loop-free MSCs. Consequently the *runs* of a system model closed by a CDL context are necessarily finite. We then extend each *run* of $[\![C|(s,\mathcal{S})]\!]$ by a specific terminal event end_C allowing the observer to catch the ending of a scenario and liveness properties to be checked.

V. OBP TOOLSET

To carry out our experiments, we used our OBP[6] tool (Figure 8). OBP is an implementation of a CDL language translation in terms of formal languages, i.e. currently FIACRE [FGP+08]. As depicted in Figure 8, OBP leverages existing academic model checkers such as TINA [BRV04] or simulators such as our explorer called 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. To do so, OBP generates either an observer automaton

[6]OBP$_t$ (OBP for TINA) is available on www.obpcdl.org

[HLR93] from each property for OBP Explorer, or SELT logic formula [BRV04] for the TINA model checker. 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 TopCased[7] or Omega[8] projects to generate FIACRE programs.

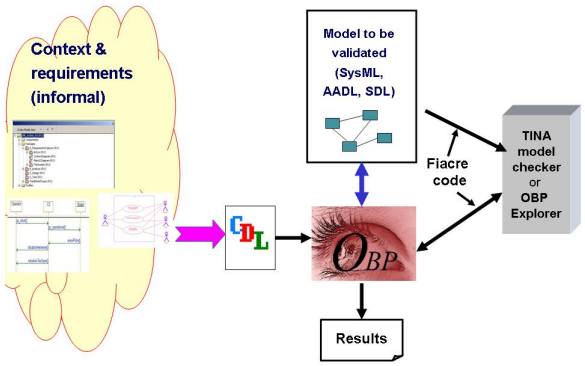

Figure 8. CDL model transformation with OBP

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 II
Exploration with TINA explorer with context splitting using OBP$_t$

N.of devices	Exploration time (sec)	N.of sub-contexts	N.of LTS config.	N.of LTS trans.
1	11	3	16 884	82 855
2	26	3	66 255	320 802
3	92	3	270 095	1 298 401
4	121	3	939 807	4 507 051
5	240	3	2 616 502	12 698 620
6	2161	40	32 064 058	157 361 783
7	4 518	55	64 746 500	322 838 592

Table II shows the amount of TINA exploration[9] for CDL examples with the use of context splitting. The first column depicts the number n of Dev asking for login to the S_CP. 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. Table II also shows the number of contexts split by OBP. For example, with 7 devices, we needed to split the CDL context in 55 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.

[7]www.topcased.org
[8]www-Omega.imag.fr
[9]Tests with same computer as for Table I

VII. Discussion and future work

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 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

[AH01] Luca De Alfaro and Thomas A. Henzinger. Interface automata. In *Proceedings of the Ninth Annual Symposium on Foundations of Software Engineering (FSE), ACM*, pages 109–120. Press, 2001.

[BH05] Dragan Bosnacki and Gerard J. Holzmann. Improving spin's partial-order reduction for breadth-first search. In *SPIN*, pages 91–105, 2005.

[BRV04] B. Berthomieu, P.-O. Ribet, and F. Verdanat. The tool TINA - Construction of Abstract State Spaces for Petri Nets and Time Petri Nets. *International Journal of Production Research*, 42, 2004.

[CES86] E.M. Clarke, E.A. Emerson, and A.P. Sistla. Automatic verification of finite-state concurrent systems using temporal logic specifications. *ACM Trans. Program. Lang. Syst.*, 8(2):244–263, 1986.

[CLM99] E. M. Clarke, D. E. Long, and K. L. Mcmillan. Compositional model checking. MIT Press, 1999.

[DPC+09] Philippe Dhaussy, Pierre-Yves Pillain, Stephen Creff, Amine Raji, Yves Le Traon, and Benoit Baudry. Evaluating context descriptions and property definition patterns for software formal validation. In Bran Selic Andy Schuerr, editor, *12th IEEE/ACM conf. Model Driven Engineering Languages and Systems (Models'09)*, volume LNCS 5795, pages 438–452. Springer-Verlag, 2009.

[DR11] Philippe Dhaussy and Jean-Charles Roger. Cdl (context description language) : Syntax and semantics. Technical report, ENSTA-Bretagne, 2011.

[FGP+08] Patrick Farail, Pierre Gaufillet, Florent Peres, Jean-Paul Bodeveix, Mamoun Filali, Bernard Berthomieu, Saad Rodrigo, Francois Vernadat, Hubert Garavel, and Frédéric Lang. FIACRE: an intermediate language for model verification in the TOPCASED environment. In *European Congress on Embedded Real-Time Software (ERTS), Toulouse, 29/01/2008-01/02/2008*. SEE, janvier 2008.

[FQ03] Cormac Flanagan and Shaz Qadeer. Thread-modular model checking. In *SPIN'03*, 2003.

[God95] P. Godefroid. The Ulg partial-order package for SPIN. *SPIN Workshop*, 1995.

[HLR93] N. Halbwachs, F. Lagnier, and P. Raymond. Synchronous observers and the verification of reactive systems. In M. Nivat, C. Rattray, T. Rus, and G. Scollo, editors, *Third Int. Conf. on Algebraic Methodology and Software Technology, AMAST'93*, Twente, June 1993. Workshops in Computing, Springer Verlag.

[Hol97] G.J. Holzmann. The model checker SPIN. *Software Engineering*, 23(5):279–295, 1997.

[HP94] G.J. Holzmann and Doron Peled. An improvement in formal verification. In *Proc. Formal Description Techniques, FORTE94*, pages 197–211, Berne, Switzerland, October 1994. Chapman & Hall.

[LPY97] Kim Guldstrand Larsen, Paul Pettersson, and Wang Yi. UPPAAL in a nutshell. *International Journal on Software Tools for Technology Transfer*, 1(1-2):134–152, 1997.

[Pel94] D. Peled. Combining Partial-Order Reductions with On-the-fly Model-Checking. In *CAV '94: Proceedings of the 6th International Conference on Computer Aided Verification*, pages 377–390, London, UK, 1994. Springer-Verlag.

[PK06] Sachoun Park and Gihwon Kwon. Avoidance of state explosion using dependency analysis in model checking control flow model. In *ICCSA (5)*, pages 905–911, 2006.

[TD03] Oksana Tkachuk and Matthew B. Dwyer. Automated environment generation for software model checking. In *In Proceedings of the 18th International Conference on Automated Software Engineering*, pages 116–129, 2003.

[Val91] Antti Valmari. Stubborn sets for reduced state space generation. In *Proceedings of the 10th International Conference on Applications and Theory of Petri Nets*, pages 491–515, London, UK, 1991. Springer-Verlag.

[Whi06] Jon Whittle. Specifying precise use cases with use case charts. In *MoDELS'06, Satellite Events*, pages 290–301, 2006.